

Concepção e Análise de Algoritmos

Técnicas I

1

Técnicas de Concepção de Algoritmos (1^a parte): programação dinâmica

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
CAL, MIEIC, FEUP
Fevereiro de 2011

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

2

Programação dinâmica (*dynamic programming*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

3

Aplicabilidade e abordagem

- ◆ Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares)
- ◆ ... Mas em que a resolução recursiva directa duplicaria trabalho (resolução repetida do mesmo subproblema)
- ◆ Abordagem:
 - 1º) Economizar tempo (evitar repetir trabalho), memorizando as soluções parciais dos sub-problemas (gastando memória!)
 - 2º) Economizar memória, resolvendo sub-problemas por ordem que minimiza nº de soluções parciais a memorizar (*bottom-up*, começando pelos casos base)
- ◆ Termo “Programação” vem da Investigação Operacional, no sentido de “formular restrições ao problema que tornam um método aplicável”

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

4

Exemplo: nC_k , versão recursiva

```
int comb(int n, int k) {
    if (k == 0 || k == n)
        return 1;
    else
        return comb(n-1, k) + comb(n-1, k-1);
}
```

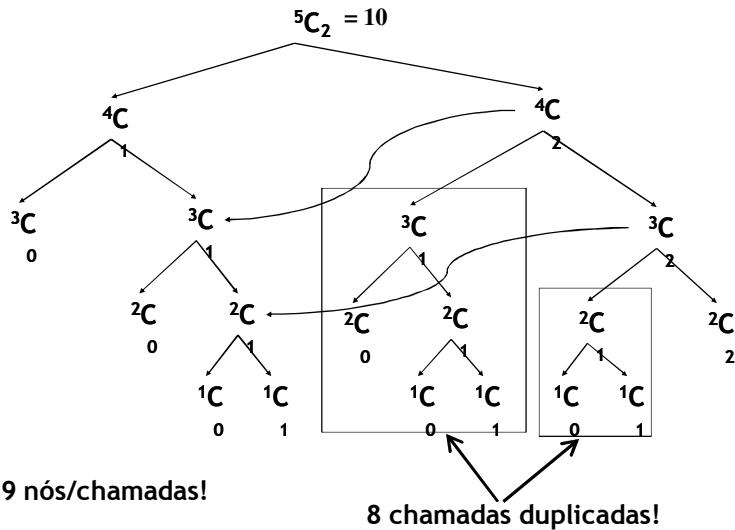
- Executa ${}^nC_{k-1}$ vezes (nº de somas a efectuar é nº de parcelas -1)
- Executa nC_k vezes (nº de 1's / parcelas que é preciso somar ...)
- Executa 2^nC_{k-1} vezes para calcular nC_k !!

Pode-se melhorar muito, evitando repetição de trabalho
(cálculos intermédios iC_j)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

5

nC_k - repetição de trabalho



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

6

nC_k - Programação dinâmica

Memorização de soluções parciais:

nC_k	k=0	k=1	K=2	K=3	K=4	k=5
n=0	1					
n=1	1	1				
n=2	1	2	1			
n=3	1	3	3	1		
n=4	1	4	6	4	1	
n=5	1	5	10	10	5	1

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

7

Implementação

Guardar apenas uma coluna, e calcular da esq. para dir.
(também se podia guardar 1 linha e calc. cima p/ baixo):

```
static final int MAXN = 50;
static int c[] = new int[MAXN+1];

int comb(int n, int k) {
    int maxj = n - k;
    for (int j = 0; j <= maxj; j++)
        c[j] = 1; n-k+1 vezes
    for (int i = 1; i <= k; i++)
        for (int j = 1; j <= maxj; j++)
            c[j] += c[j-1]; k(n-k) vezes
    return c[maxj];
}
```

$T(n,k) = O(k(n-k))$
 $S(n,k) = O(n-k)$

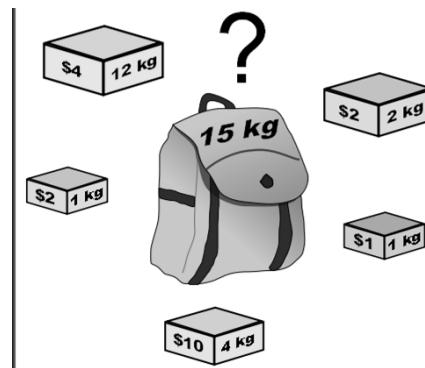
Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

8

Problema da mochila

- ◆ Um ladrão encontra o cofre cheio de itens de vários tamanhos e valores, mas tem apenas uma mochila de capacidade limitada; qual a combinação de itens que deve levar para maximizar o valor do roubo?

- Tamanhos e capacidades inteiros
- Vamos assumir nº ilimitado de itens de cada tipo



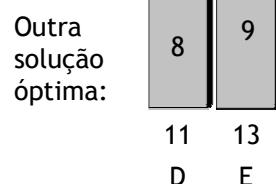
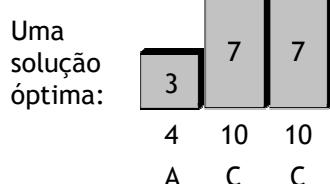
Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

9

Exemplo

Itens	
Tamanho	
Valor	
Nome	
3	A
4	B
7	C
8	D
9	E

Capacidade da mochila: 17



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

10

Estratégia de prog. dinâmica

- ◆ Calcular a melhor combinação para todas as mochilas de capacidade 1 até M (capacidade pretendida)
- ◆ Começar por considerar que só se pode usar o item 1, depois os itens 1 e 2, etc., e finalmente todos os itens de 1 a N ($N = \text{nº de itens}$)
- ◆ Cálculo é eficiente em tempo e espaço se efectuado pela ordem apropriada

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

11

Dados

- ◆ Entradas:
 - N - nº de itens (com nº de cópias ilimitado de cada item)
 - $\text{size}[i]$ ($1 \leq i \leq N$) - tamanho (inteiro) do item i
 - $\text{val}[i]$ ($1 \leq i \leq N$) - valor do item i
 - M - capacidade da mochila (inteiro)
- ◆ Dados de trabalho, no final de cada iteração i (de 0 a N)
 - $\text{cost}[k]$ ($1 \leq k \leq M$) - melhor valor que se consegue com mochila de capacidade k , usando apenas itens de 1 a i
 - $\text{best}[k]$ ($1 \leq k \leq M$) - último item seleccionado p/ obter melhor valor com mochila de capac. k , usando apenas itens de 1 a i
- ◆ Dados de saída:
 - $\text{cost}[M]$ - melhor valor que se consegue c/ mochila de cap. M
 - $\text{best}[M], \text{best}[M-\text{size}[\text{best}[M]]], \text{etc.}$ - itens seleccionados

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

12

Formulação recursiva

- ◆ Caso base ($i = 0; k = 1, \dots, M$):
- | |
|----------------------------|
| $\text{cost}[k]^{(0)} = 0$ |
| $\text{best}[k]^{(0)} = 0$ |

- ◆ Caso recursivo ($i = 1, \dots, N; k = 1, \dots, M$) :

$$\begin{aligned} \text{cost}[k]^{(i)} &= \begin{cases} \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)}, & \text{se } \begin{cases} \text{size}[i] \leq k \\ \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)} > \text{cost}[k]^{(i-1)} \end{cases} \\ \text{cost}[k]^{(i-1)}, & \text{no caso contrário} \end{cases} \\ \text{best}[k]^{(i)} &= \begin{cases} i, & \text{no primeiro caso acima (usa o item } i \text{)} \\ \text{best}[k]^{(i-1)}, & \text{no segundo caso acima (não usa o item } i \text{)} \end{cases} \end{aligned}$$

Encher o resto

Permite usar repetidamente o item i
(senão, escrevíamos $i-1$)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

Codificação

Tempo: $T(N,M) = O(NM)$

Espaço: $S(N,M) = O(M)$

```

int[] cost = new int[M+1]; // iniciado c/ 0's
int[] best = new int[M+1]; // iniciado c/ 0's

for (int i = 1; i <= N; i++) {
    for (int k = size[i]; k <= M; k++)
        if (val[i] + cost[k-size[i]] > cost[k]) {
            cost[k] = val[i] + cost[k-size[i]];
            best[k] = i;
        }
    } Como k é percorrido por ordem crescente  
cost[k-size[i]] já tem o valor da iteração i
// impressão de resultados (valor e itens)
print(cost[M]);
for (int k = M; k > 0; k -= size[best[k]])
    print(best[k]);

```

Como k é percorrido por ordem crescente
cost[k-size[i]] já tem o valor da iteração i

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

***Evolução dos dados de trabalho**

i	size	val	k	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
0	-	-	cost[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			best[k]	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	cost[k]	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			best[k]	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	cost[k]	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
			best[k]	0	0	0	1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
3	7	10	cost[k]	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
			best[k]	0	0	0	1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
4	8	11	cost[k]	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
			best[k]	0	0	0	1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
5	9	13	cost[k]	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
			best[k]	0	0	0	<u>1</u>	2	2	1	3	4	5	<u>3</u>	3	5	3	3	4	5	<u>3</u>

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

15

* Subsequência crescente mais comprida

- ◆ Exemplo:

- Sequência $S = (9, 5, 2, 8, 7, 3, 1, 6, 4)$
- Subsequência crescente mais comprida (elem's não necessariamente contíguos): $(2, 3, 4)$ ou $(2, 3, 6)$

- ◆ Formulação:

- s_1, \dots, s_n - sequência
- l_i - compr. da maior subseq. crescente de (s_1, \dots, s_i)
- p_i - predecessor de s_i nessa subsequência crescente
- $l_i = 1 + \max \{ l_k \mid 0 < k < i \wedge s_k < s_i \}$ ($\max \{ \} = 0$)
- p_i = valor de k escolhido para o máx. na expr. de l_i
- Comprimento final: $\max(l_i)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

16

* Cálculo para o exemplo dado

	i	1	2	3	4	5	6	7	8	9
Sequência	s_i	9	5	<u>2</u>	8	7	<u>3</u>	1	<u>6</u>	4
Tamanho	l_i	1	1	1	2	2	2	1	<u>3</u>	3
Predecessor	p_i	-	-	-	2	2	<u>3</u>	-	<u>6</u>	6

Resposta: $(2, 3, 6)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

17

* Números de Fibonacci

- ◆ Formulação recursiva:
 - $F(0) = 0$
 - $F(1) = 1$
 - $F(n) = F(n-1) + F(n-2)$, $n > 1$
- ◆ Para calcular $F(n)$, basta memorizar os dois últimos elementos da sequência para calcular o seguinte:

```
int Fib(int n) {
    int a = 1, b = 0 ; // F(-1), F(0)
    for (int i=1; i <= n; i++) {int t = a; a = b; b += t; }
    return b;
}
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

18

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992
- ◆ Slides de Maria Cristina Ribeiro

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

1

Técnicas de Concepção de Algoritmos (1^a parte): algoritmos gananciosos

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Fevereiro de 2011

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

2

Algoritmos gananciosos (*greedy algorithms*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

3

Algoritmos Gananciosos

- ◆ É qualquer algoritmo que aplica uma heurística de solução em que se tenta aplicar uma escolha óptima local em todo e cada estágio da solução.
- ◆ Aplicável a problemas de optimização (*maximização* ou *minimização*)
- ◆ Em diversos problemas, a optimização local garante também a optimização global, permitindo encontrar a solução óptima de forma eficiente
- ◆ **Subestrutura óptima:** um problema tem subestrutura óptima se uma solução óptima p/ problema contém soluções óptimas para os seus subproblemas!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

4

Estratégia Gananciosa

- ◆ Um algoritmo ganancioso funciona em fases. Em cada fase verifica-se a seguinte estratégia:
 1. Pega-se o melhor que se pode obter no exacto momento, sem considerar as consequências futuras ou para o resultado final
 2. Por se ter escolhido um **óptimo local** a cada passo, espera-se por acabar a encontrar um **óptimo global**!
- ◆ Portanto, a escolha que parece se a melhor no momento é a opção escolhida! Assim,
 - Prova-se que quando há uma escolha a fazer, uma das escolhas possíveis é a “gananciosa.” Portanto, é sempre seguro optar-se por esta escolha
 - Demonstra-se que todos os subproblemas resultantes de uma alternativa gananciosa são vazios, excepto um (o resultado)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

5

Premissas

- ◆ Cinco principais características que suportam essa solução:
 1. Um **conjunto de candidatos**, de onde a solução é criada
 2. Uma **função de selecção**, que escolhe o melhor candidato a ser incluído na solução
 3. Uma **função de viabilidade**, que determina se o candidato poderá ou não fazer parte da solução
 4. Uma **função objectivo**, que atribui um valor a uma solução, ou solução parcial
 5. Uma **função solução**, que determinará se e quando se terá chegado à solução completa do problema

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

6

Algoritmo abstracto

- ◆ Inicialmente o conjunto de itens está vazio (i.e. conjunto solução)
- ◆ A cada passo:
 - Um item será adicionado ao conjunto solução, pela função de selecção
 - SE o conjunto solução tornar-se inviável, ENTÃO rejeita-se os itens em consideração (não voltando a seleccioná-los)
 - SENÃO o conjunto solução ainda é viável, ENTÃO adiciona-se os itens a serem considerados

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

7

Problema do troco



extrair 8 cêntimos

(com nº mínimo de moedas)

Saco / depósito / stock de moedas

$\text{extrair}(8, \{1, 1, 1, 2, 2, 2, 2, 5, 5\})$

(com nº mínimo de moedas)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

8

Resol. c/ algoritmo ganancioso

$\text{extrair}(8, \{1, 1, 1, 2, 2, 2, 2, 5, 5\})$

5

$\text{extrair}(3, \{1, 1, 1, 2, 2, 2, 2, 5\})$

2

$\text{extrair}(1, \{1, 1, 1, 2, 2, 2, 5\})$

1

$\text{extrair}(0, \{1, 1, 2, 2, 2, 5\})$

FIM

Escolhe-se a moeda de valor mais alto que não excede o montante em falta (pois com moedas de valor mais alto o nº de moedas necessário será mais baixo)

Sub-problema do mesmo tipo

Dá a solução óptima, se o sistema de moedas tiver sido concebido apropriadamente (caso do euro) e não existirem problemas de stock!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

9

Implementação iterativa

```

static final int moedas[] = {1,2,5,10,20,50,100,200};

// stock[i] = nº de moedas de valor moedas[i]
public int[] select(int montante, int[] stock) {
    int[] sel = new int[moedas.length];
    for (int i=moedas.length-1; montante>0 && i>=0; i--)
        if (stock[i] > 0 && moedas[i] <= montante) {
            int n_moed=Math.min(stock[i],montante/moedas[i]);
            sel[i] += n_moed;
            montante -= n_moed * moedas[i];
        }
    if (montante > 0)
        return null;
    else
        return sel;
}

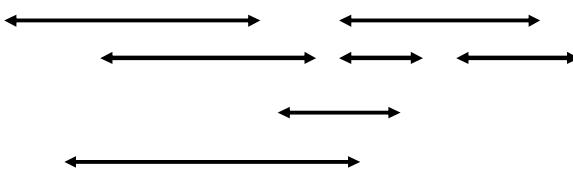
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

10

Escalonamento de actividades

- ◆ Problema: dado um conjunto de actividades, encontrar um subconjunto com o maior número de actividades não sobrepostas!
- ◆ Input: Conjunto S de n actividades, a_1, a_2, \dots, a_n .
 - > s_i = instante de início da actividade i .
 - > f_i = instante de fim da actividade i .
- ◆ Output: Subconjunto A de número máximo de actividades compatíveis (i.e. não sobrepostas)



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

11

Escalonamento de actividades

Subestrutura óptima:

- ◆ Assume-se que as actividades estão ordenadas.
 - $f_1 \leq f_2 \leq \dots \leq f_n$.
- ◆ Supondo-se q/ uma solução óptima inclua actividade a_k .
 - Isso gera dois subproblemas:
 - Seleccionar de a_1, \dots, a_{k-1} , actividades compatíveis entre si, e que terminam antes de a_k começar (compatíveis com a_k).
 - Seleccionar de a_{k+1}, \dots, a_n , actividades compatíveis entre si, e que iniciam depois de a_k terminar.
 - A solução para os dois subproblemas deve ser óptima.
 - * Fica como exercício provar esta condição!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

12

Escalonamento de actividades

Abordagem recursiva:

- ◆ Seja S_{ij} = subconjunto de actividades em S q/ iniciam depois de a_i terminar e terminam antes de a_j começar.
- ◆ Subproblemas: Seleccionar o máximo número de actividades mutuamente compatíveis de S_{ij} .
- ◆ Seja $c[i, j]$ = tamanho do subconjunto de tamanho máximo de actividades mutuamente compatíveis em S_{ij} .

Solução recursiva

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset \\ \max_{i < k < j} \{c[i, k] + c[k, j] + 1\} & \text{if } S_{ij} \neq \emptyset \end{cases}$$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

13

* Escalonamento de actividades

```
// event-schedule
// schedule as many non-conflicting events as possible
function event-schedule(events array of s[1..n], j[1..n]): set
    if n == 0: return fi
    if n == 1: return {events[1]} fi
    let event := events[1]
    let S1 := union(event-schedule(events -
                                    set of conflicting events), event)
    let S2 := event-schedule(events - {event})
    if S1.size() >= S2.size():
        return S1
    else
        return S2
    fi
end
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

14

Escalonamento de actividades

Abordagem gananciosa:

- ◆ Considerar as actividades numa ordem específica
- ◆ Escolher a “melhor opção” de actividade
- ◆ Descartar todas as actividades incompatíveis com a actividade seleccionada
- ◆ Estratégias
 - “Earliest starting time” -> ascendente em Si
 - “Earliest finishing time” -> ascendente em Fi
 - “Shortest interval” -> ascendente em Fi - Si
 - “Fewest conflicts” -> para cada actividade, contar o número de conflitos e ordenar segundo este número.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

15

Escalonamento de actividades

Abordagem gananciosa:

$$A = \{a_1, a_2, \dots, a_i, \dots, a_n\}$$

$$R = \emptyset$$

While $S \neq \emptyset$

$a \leftarrow a_i \mid \text{smallest finishing time}$

$R \leftarrow R \cup \{a\}$

$A \leftarrow A - \forall a_j \mid a_j \text{ não é compatível com } a_i$

EndWhile

Return R

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

16

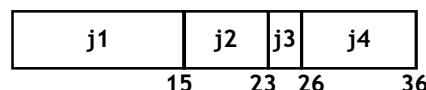
* Escalonamento de actividades

Variação do problema de escalonamento de actividades:

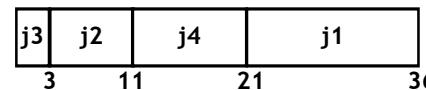
- ◆ Dados: tarefas (*jobs*) e tempo (duração)
- ◆ Objectivo: sequenciar tarefas minimizando o tempo médio de conclusão
- ◆ Método: tarefas mais curtas (que acabam mais cedo) primeiro

Tarefa	Tempo
j1	15
j2	8
j3	3
j4	10

Tempo médio: 25



Tempo médio: 17.75



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

17

*Demonstração de optimalidade

- ◆ Tarefas: $j_{i1}, j_{i2}, \dots, j_{in}$ (ordenadas)
- ◆ Durações: $t_{i1}, t_{i2}, \dots, t_{in}$
- ◆ Instantes de conclusão: $t_{i1}, t_{i1}+t_{i2}, \dots$
- ◆ Custo total da solução

$$\sum_{k=1}^n (n - k + 1) t_{ik} = (n+1) \sum_{k=1}^n t_{ik} - \sum_{k=1}^n k t_{ik}$$

- ◆ Se existe $x > y$ tal que $t_{ix} < t_{iy}$, troca de j_{ix} e j_{iy} diminui custo

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

18

Outros Exemplos de Problemas

- ◆ Problemas em que garante uma solução óptima:
 - Problema do troco, desde que não haja falta de stock
 - Problema de escalonamento
 - Árvores de expansão mínima
 - Codificação de Huffman
 - Dijkstra, para cálculo do caminho mais curto num grafo
- ◆ Problemas em que não garante uma solução óptima
 - Problema da mochila (mas pode dar boas aproximações ...)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992
- ◆ Slides de Maria Cristina Ribeiro

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

1

Técnicas de Concepção de Algoritmos (1^a parte): algoritmos de retrocesso

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Fevereiro de 2011

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

2

Para pensar...

- ◆ “Theory is when you know something, but it doesn’t work.
Practice is when something works, but you don’t know why.
Programmers combine theory and practice: Nothing works and they don’t know why.”

(unknown)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

3

Algoritmos de retrocesso (*backtracking*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

4

Algoritmos de retrocesso

- ◆ Um dado problema tem um conjunto de restrições e possivelmente uma função objectivo
- ◆ Uma solução optimiza a função objectivo e/ou a satisfaz
- ◆ Pode-se representar o espaço de solução para o problema utilizando-se uma árvore de espaço de estados
 - A raiz da árvore representa 0 escolhas
 - Nós ao nível 1 representam primeira escolha
 - Nós ao nível 2 representam segunda escolha, etc...
- ◆ O caminho da raiz a uma folha representa uma solução candidata

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

5

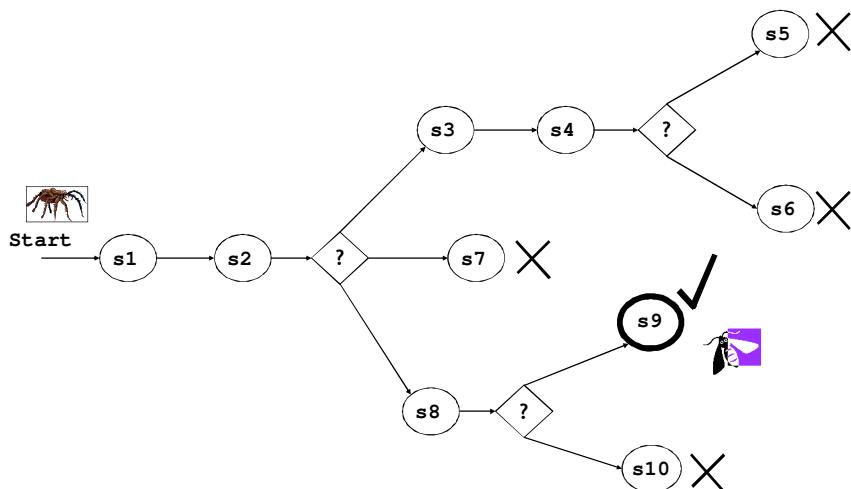
Algoritmos de retrocesso

- ◆ Algoritmos de *tentativa e erro*
- ◆ Contexto geral de aplicação:
 - Explorar um *espaço de estados* à procura dum *estado-objectivo*
 - Estado = estado de jogo, sub-problema a resolver, posição, etc.
 - Sem algoritmos eficientes que levem directamente ao objectivo
- ◆ Estratégia:
 - Ao chegar a um *ponto de escolha* (c/ vários estados seguintes), escolher uma das opções e prosseguir a exploração
 - Chegando a um “beco sem saída”, *retroceder* até ao ponto de escolha + próximo c/alternativas p/explorar, e tentar outra alt.
- ◆ Exemplos:
 - Problema do troco quando há falta de stock de algumas moedas
 - Sudoku, 8 rainhas, *puzzles* em geral.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

6

Ilustração



Complicações: ciclos, caminhos paralelos

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

7

Implementação recursiva

- ◆ Implementado normalmente de forma recursiva
 - avanço corresponde a uma chamada recursiva
 - retrocesso corresponde ao retorno de chamadas recursivas

Explore state/node N:

1. if N is a goal state/node, return “success”
2. (optional) if N is a leaf state/node, return “failure”
3. for each successor/child C of N,
 - 3.1. (if appropriate) set new state
 - 3.2. explore state/node C
 - 3.3. if exploration was successful, return “success”
 - 3.4. (if step 3.1 was performed) restore previous state
4. return “failure”

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

8

Ex. Soma de subconjuntos

- ◆ Problema: dados n positivos inteiros w_1, \dots, w_n e um inteiro positivo S , encontrar todos os subconjuntos de w_1, \dots, w_n cuja soma é S
- ◆ Exemplo: $n = 3$, $S = 6$, $W = \{2, 4, 6\}$
- ◆ Solução:
 - $\{2, 4\}$
 - $\{6\}$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

9

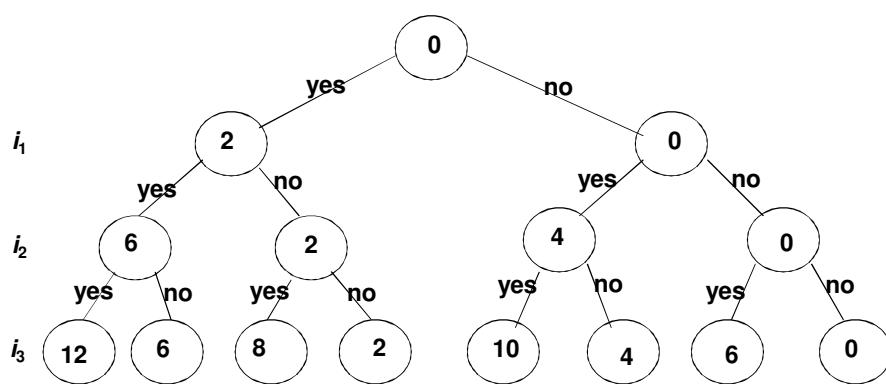
Ex. Soma de subconjuntos

- ◆ Para este caso, assume-se uma árvore binária para o espaço de estados
- ◆ Nós ao nível 1 representam incluir (sim ou não) o item 1, nós ao nível 2 representam incluir item 2, etc...
- ◆ O ramo esquerdo da árvore inclui w_1 enquanto o ramo direito da árvore exclui w_1
- ◆ Os nós contêm as somas dos pesos incluídos até então!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

10

Problema: soma de subconjuntos
Árvore de espaço de estados para 3 itens
 $w_1 = 2, w_2 = 4, w_3 = 6$ e $S = 6$



A soma dos inteiros incluídos é guardada nos nós!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

11

Ex. Soma de subconjuntos

- ◆ Problemas como este podem ser resolvidos realizando-se uma pesquisa/busca em profundidade
- ◆ Cada nó guardará o seu nível (profundidade) e a sua solução (possivelmente parcial) corrente
- ◆ Uma busca em profundidade pode verificar se um nó v é uma folha:
 - Se v é uma folha, então verifica-se se a solução corrente satisfaz as restrições do problema
 - Extensões a este método podem ser implementadas a fim de se encontrar um solução óptima

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

12

Ex. Soma de subconjuntos

- ◆ Uma estratégia baseada unicamente em busca/pesquisa em profundidade pode representar uma alternativa muito cara em termos de tempo de processamento!
- ◆ Neste caso, não se verifica para todo estado solução (nó) quando a solução foi alcançada, ou mesmo se uma solução parcial poderá levar a uma solução satisfatória
- ◆ Questão: é possível desenvolver uma alternativa mais eficiente?

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

13

Estratégia de Retrocesso

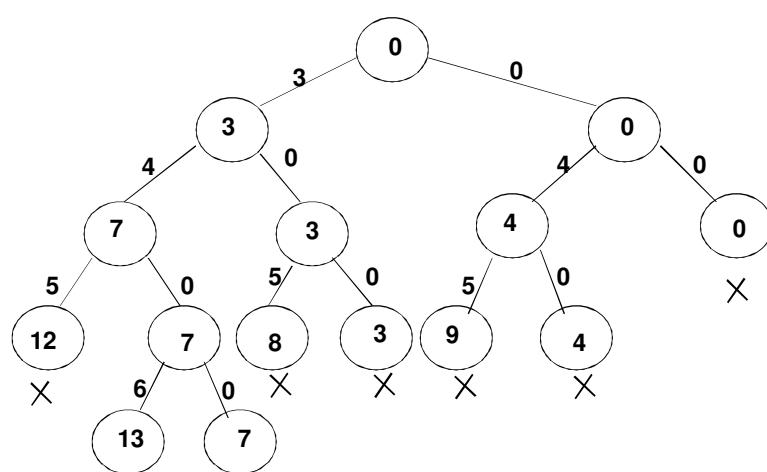
- ◆ Definição: chama-se a um nó “não promissor” caso este não conduza a uma solução viável (ou óptima). Caso contrário, este será tido como um nó “promissor”
- ◆ Ideia básica: retrocesso consiste em realizar uma pesquisa em profundidade na árvore de espaço de estados, verificando se um nó é promissor, e caso o nó não seja promissor, retroceder até o nó pai.
- ◆ A uma árvore de espaço de estados que contém apenas nós expandidos chama-se árvore de espaço de estados podada

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

14

Árvore podada de espaço de estados (p/ encontrar todas as soluções)

$$w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$$



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

15

Estratégia de Retrocesso

```

void checknode (node v) {
    node u

    if ( promising ( v ) )
        if ( aSolutionAt( v ) )
            write the solution
        else //expand the node
            for ( each child u of v )
                checknode ( u )

}

```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

16

Estratégia de Retrocesso

- ◆ *Checknode* usa as funções:
 - *promissing(v)* que verifica se a solução parcial representada pelo nó *v* poderá levar à solução desejada
 - *aSolutionAt(v)* que verifica se a solução parcial representada pelo nó *v* resolve o problema em questão

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

17

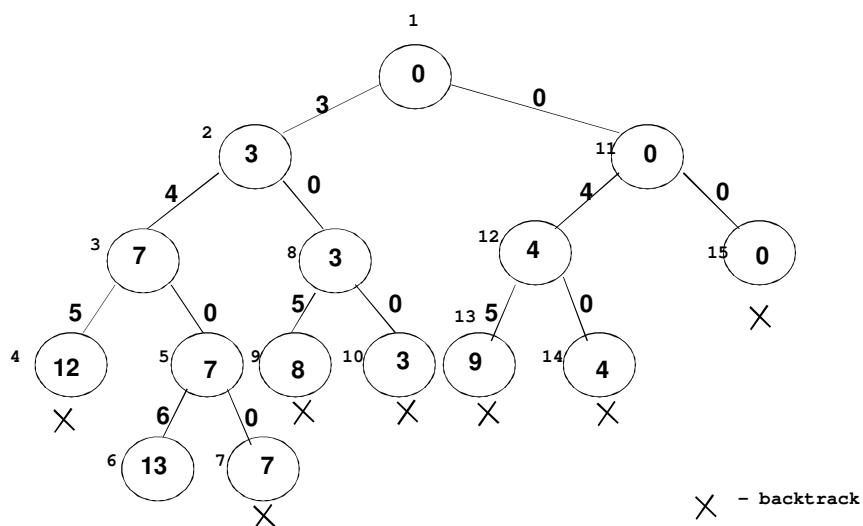
Estratégia de Retrocesso

- ◆ Quando um nó é “promissor”?
- Considere um nó ao nível i :
- $weightSoFar$: peso do nó, i.e. soma dos números incluídos na solução parcial que o nó representa
 - $totalPossibleLeft$: peso dos itens remanescentes ($i + 1$ a n) para um nó ao nível i
 - Um nó ao nível i é “não promissor” se
 $weightSoFar + totalPossibleLeft < S$ (ou)
 $weightSoFar + w[i + 1] > S$
 - Para se poder utilizar a função *promissing*, os elementos w_i devem estar ordenados numa ordem não decrescente!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

18

Árvore podada de espaço de estados
 $w_1 = 3, w_2 = 4, w_3 = 5, w_4 = 6; S = 13$



Nodes numbered in “call” order

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

19

```

sumOfSubsets ( i, weightSoFar, totalPossibleLeft )
  1) if (promising ( i ))           //may lead to solution
  2) then if ( weightSoFar == S )
  3)   then print include[ 1 ] to include[ i ] //found solution
  4) else    //expand the node when weightSoFar < S
  5)   include [ i + 1 ] = "yes"          //try including
  6)   sumOfSubsets ( i + 1, weightSoFar + w[i + 1],
                     totalPossibleLeft - w[i + 1] )
  7)   include [ i + 1 ] = "no"           //try excluding
  8)   sumOfSubsets ( i + 1, weightSoFar ,
                     totalPossibleLeft - w[i + 1] )

boolean promising ( i )
  1) return ( weightSoFar + totalPossibleLeft  $\geq$  S ) &&
             ( weightSoFar == S || weightSoFar + w[i + 1]  $\leq$  S )

```

Prints all solutions!

Chamada inicial da função sumOfSubsets(0, 0, Σw_i)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

20

Problemas de optimização

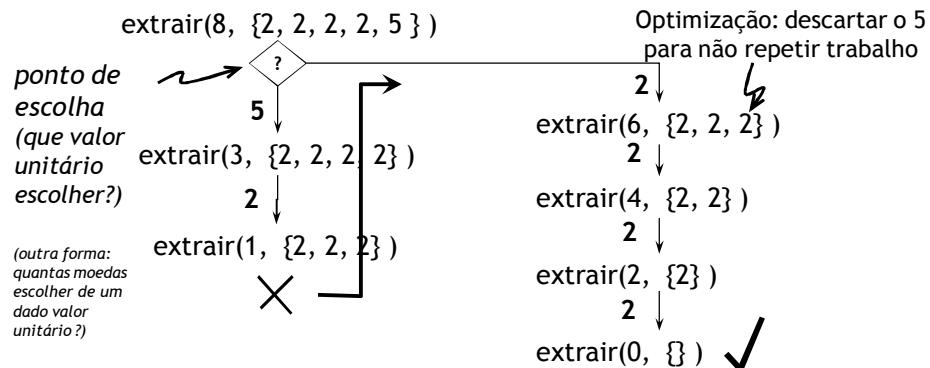
- ◆ Para problemas de optimização, considera-se também:
 - *best* - valor da melhor solução encontrada até então
 - *value(v)* - valor da solução no nó *v*
 - Deve-se modificar a função *promissing(v)*
 - *best* é inicializado com um valor igual a uma solução candidata ou pior que qualquer uma solução possível
 - *best* é actualizado com *value(v)* se a solução em *v* é “melhor”
 - Ser “melhor” dependerá do problema (*maximização ou minimização*)!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

21

Exemplo: Problema do troco

- ◆ Na falta de stock de algumas moedas, o algoritmo ganancioso pode não dar solução, quando ela existe (*)
- ◆ Resolução: retroceder até ao ponto de escolha mais próximo e escolher a moeda de valor mais baixo a seguir



(*) também pode dar uma solução não óptima, mas isso resolve-se de outra forma - com programação dinâmica

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

22

Implementação recursiva

```
static final int moedas[] = {1, 2, 5, 10, 20, 50, 100, 200};

// stock[i] = nº de moedas de valor moedas[i]
public int[] select(int montante, int[] stock) {
    int[] sel = new int[moedas.length];
    return select(montante, stock, sel, moedas.length-1) ? sel : null;
}

boolean select(int mont, int[] stock, int[] sel, int maxIdx) {
    /*1.*/ if (mont == 0)
        return true;
    /*3.*/ for (int i = maxIdx; i >= 0; i--)
        if (stock[i] > sel[i] && moedas[i] <= mont) {
            /*3.1*/ sel[i]++;
            mont -= moedas[i];
            /*3.2*/ if (select(mont, stock, sel, maxIdx))
                return true;
            /*3.3*/ sel[i]--;
            mont += moedas[i];
        }
    /*4.*/ return false;
}
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

23

Eficiência temporal

- ◆ Tempo de execução no pior caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial
 - Caso em que o montante pretendido excede o total em stock
 - Como calcular a dimensão do espaço de estados?
- ◆ No algoritmo apresentado, não há duas chamadas de *select* para o mesmo estado do array *sel*
- ◆ N° de estados possíveis de *sel* (nº de soluções potenciais a testar e nº máximo de chamadas de *select*) é:

$$\prod_{i=0, \dots, \text{moedas.length}-1} (1+stock[i]) \quad (\text{nº subconj.s de conj. c/repet.})$$
- ◆ Exemplo: $stock[i] = 9$ ($i=0, \dots, 7$), montante pretendido superior ao total em stock => 10^8 chamadas!!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

24

Poda da pesquisa

- ◆ Interromper a pesquisa em ramos que garantidamente não levam a nenhuma solução
- ◆ Exemplo no problema do troco: interromper a pesquisa (e retornar indicação de insucesso) quando o valor do stock utilizável é inferior ao montante em falta


```
boolean select(int mont, int[] stock, int[] sel, int maxIdx) {
    if (mont == 0)
        return true;
    if (mont > total(stock,0,maxIdx)-sel[maxIdx]*moedas[maxIdx])
        return false;
    ...
}
```
- ◆ Melhora o desempenho mas podem continuar a existir casos patológicos com tempo de execução exponencial

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

25

Variantes de aplicação

- ◆ Encontrar uma solução (caso estudado até aqui)
 - A pesquisa pára assim que se encontra a primeira solução
- ◆ Encontrar todas as soluções
 - Quando se encontra uma solução, processa-se essa solução (imprimir, etc.), mas não se pára a exploração
 - Retrocede-se para o ponto de escolha mais próximo como se tivéssemos chegado a um “beco sem saída”
- ◆ Encontrar a melhor solução
 - Variante de encontrar todas as soluções, em que se vai guardando a melhor solução encontrada até ao momento
 - Podar a pesquisa: interromper um caminho de pesquisa quando temos a certeza que não permite chegar a uma solução melhor

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

26

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

1

Técnicas de Concepção de Algoritmos (1^a parte): divisão e conquista

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Fevereiro de 2011

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

2

Divisão e Conquista (*divide and conquer*)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

3

Divisão e conquista

- ◆ Divisão: resolver recursivamente problemas mais pequenos (até caso base)
- ◆ Conquista: solução do problema original é formada com as soluções dos subproblemas
- ◆ Há divisão quando o algoritmo tem pelo menos 2 chamadas recursivas no corpo
- ◆ Subproblemas devem ser disjuntos
 - Senão, resolver de forma bottom-up com programação dinâmica
- ◆ Divisão em subproblemas de dimensão semelhante é importante para se obter uma boa eficiência temporal
- ◆ Algoritmos adequados para processamento paralelo

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

4

Divisão e conquista

- ◆ Dada uma instância do problema x , a técnica Divisão-e-Conquista funciona da seguinte maneira:

```
function DAQ(  $x$  )
  if  $x$  é suficientemente pequeno then
    resolver  $x$  directamente
  else
    dividir  $x$  em subinstâncias:  $x_1, \dots, x_k$ 
    for  $i := 1$  to  $k$  do  $y_i := \text{DAQ}( x_i )$ 
     $y := \sum y_i$ 
  return  $y$ 
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

5

Exemplo: cálculo de x^n

- ◆ Resolução iterativa com n multiplicações: $T(n) = O(n)$
- ◆ Resolução mais eficiente, com divisão e conquista:

$$x^n = \begin{cases} 1, & \text{se } n = 0 \\ x, & \text{se } n = 1 \\ x^{\frac{n}{2}} \times x^{\frac{n}{2}}, & \text{se } n \text{ par} > 1 \\ x \times x^{\frac{n-1}{2}} \times x^{\frac{n-1}{2}}, & \text{se } n \text{ ímpar} > 1 \end{cases}$$

```
double power(double x, int n) {
    if (n == 0) return 1;
    if (n == 1) return x;
    double p = power(x, n / 2);
    if (n % 2 == 0) return p * p;
    else return x * p * p;
}
```

- ◆ Divisão em subproblemas iguais, junção em tempo $O(1)$
- ◆ N° de multiplicações reduzido para aprox. $\log_2 n$
- ◆ $T(n) = O(\log n)$ mas $S(n) = O(\log n)$ (espaço)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

6

Exemplo: ordenação de arrays

- ◆ Mergesort
 - Ordenar 2 subsequências de igual dimensão e juntá-las
 - $T(n) = 2 T(n/2) + k n$ ($n > 1$) \Rightarrow
 - $T(n) = O(n \log n)$, tanto no pior caso como no caso médio
- ◆ Quicksort
 - Ordenar elementos menores e maiores que *pivot*, concatenar
 - $T(n) = O(n^2)$ no pior caso (1 elemento menor, restantes maiores)
 - $T(n) = O(n \log n)$ no melhor caso e no caso médio (*)
 - (*) com escolha aleatória do pivot!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

7

Exemplo: *Mergesort*

- ◆ Seja $S = \{s_1, \dots, s_n\}$ um conjunto que se pretenda ordenar. Se $S = \emptyset$ ou $S = \{s\}$, então nada é necessário!
- ◆ Dividir: remover todos os elementos de S e colocá-los em duas subsequências: S_1 e S_2 , cada uma com $\sim n/2$ elementos
- ◆ Conquistar: consiste em ordenar S_1 e S_2 , utilizando mergesort
- ◆ Combinar: colocar os elementos de volta em S , unindo as sequências ordenadas S_1 e S_2 numa sequência ordenada única.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

8

Exemplo: *Mergesort*

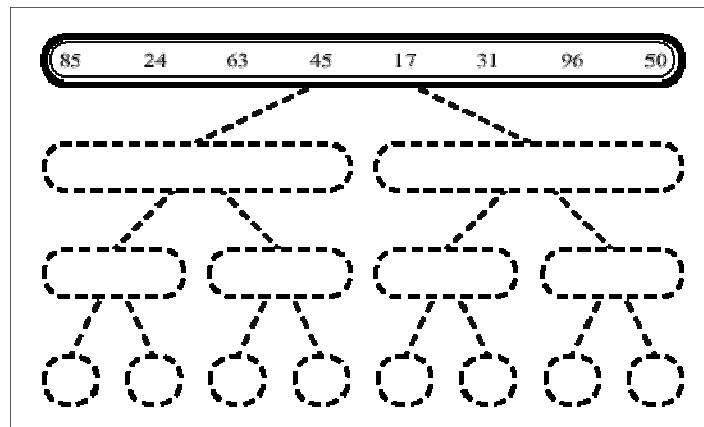
```
Merge-Sort(A, p, r)
  if p < r then
    q ← (p+r)/2
    Merge-Sort(A, p, q)
    Merge-Sort(A, q+1, r)
    Merge(A, p, q, r)
```

```
Merge(A, p, q, r)
  Take the smallest of the two topmost elements of
  sequences A[p..q] and A[q+1..r] and put into the
  resulting sequence. Repeat this, until both sequences
  are empty. Copy the resulting sequence into A[p..r].
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

9

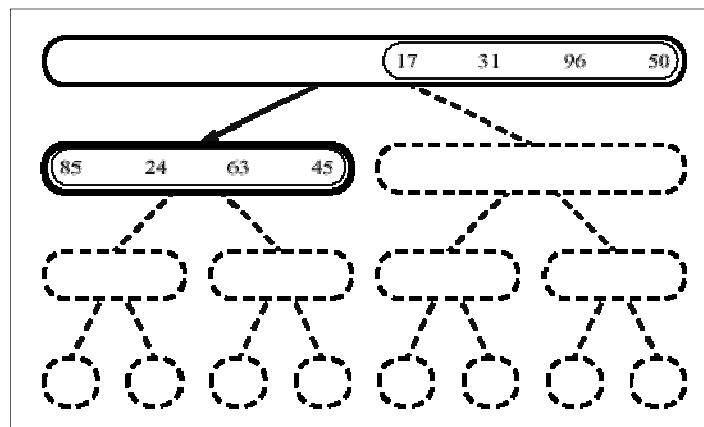
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

10

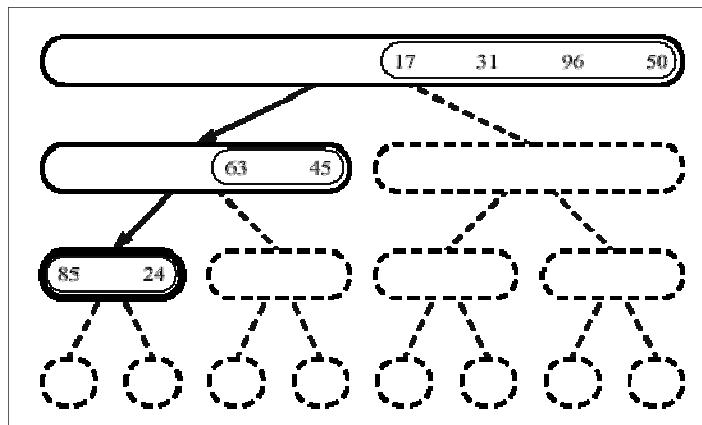
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

11

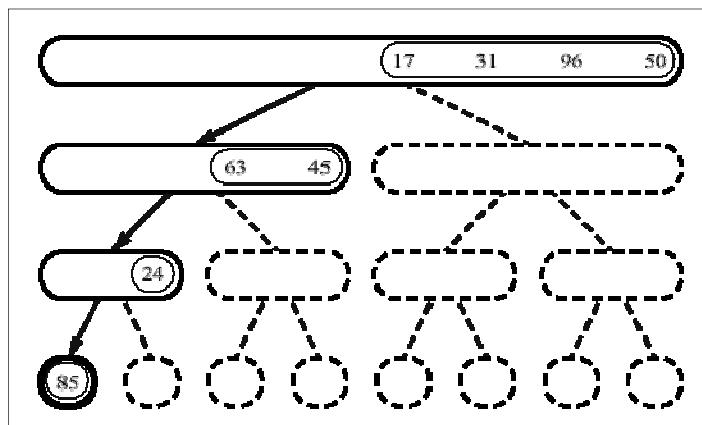
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

12

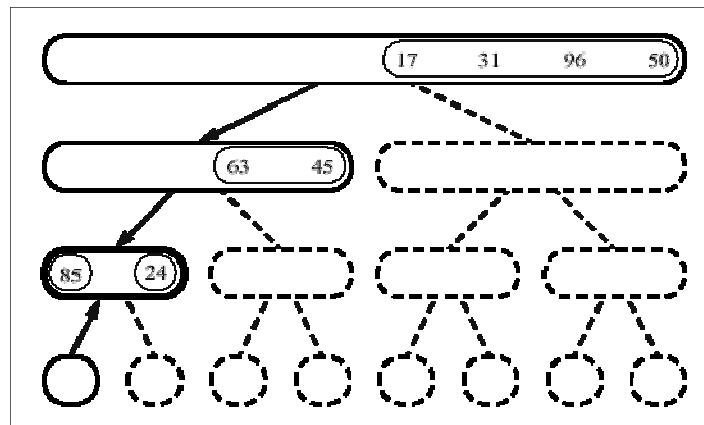
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

13

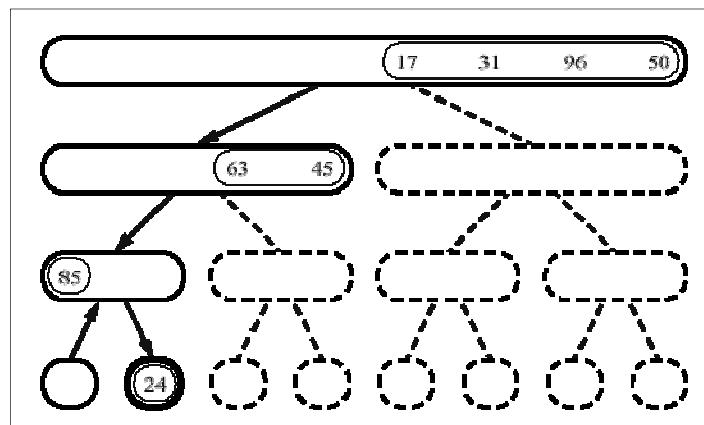
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

14

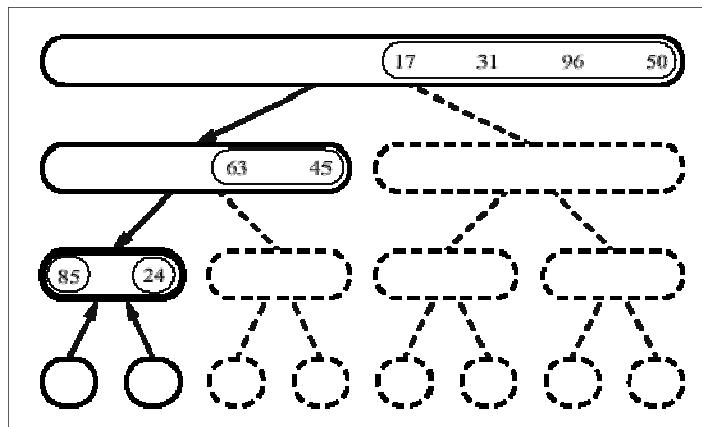
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

15

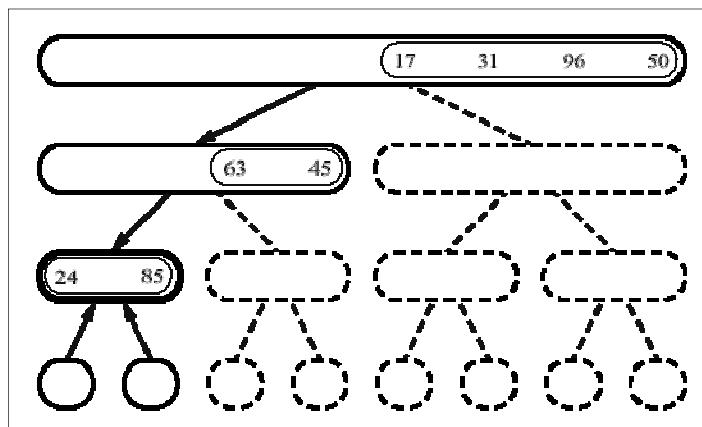
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

16

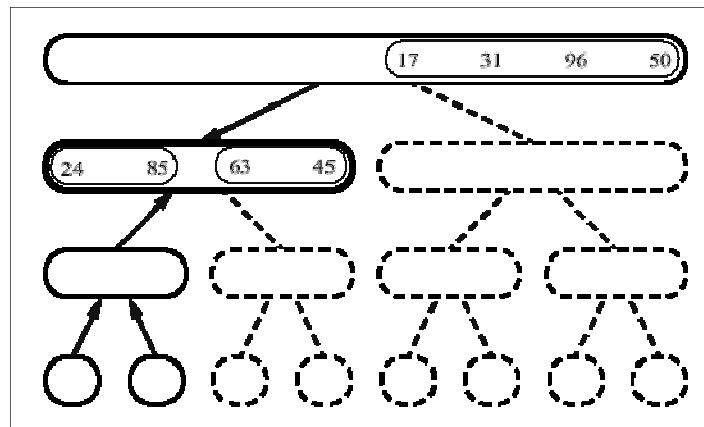
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

17

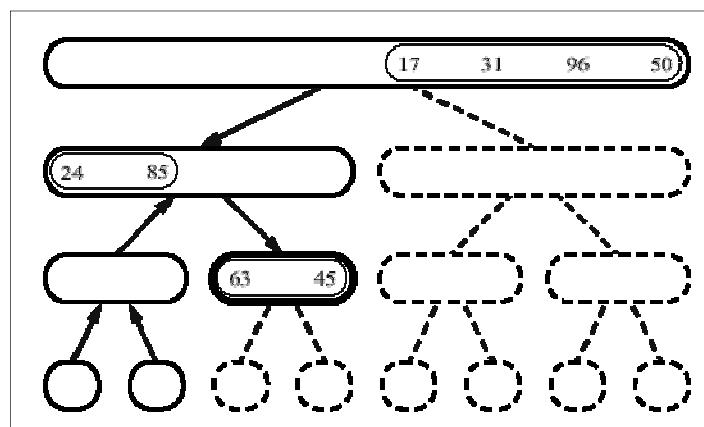
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

18

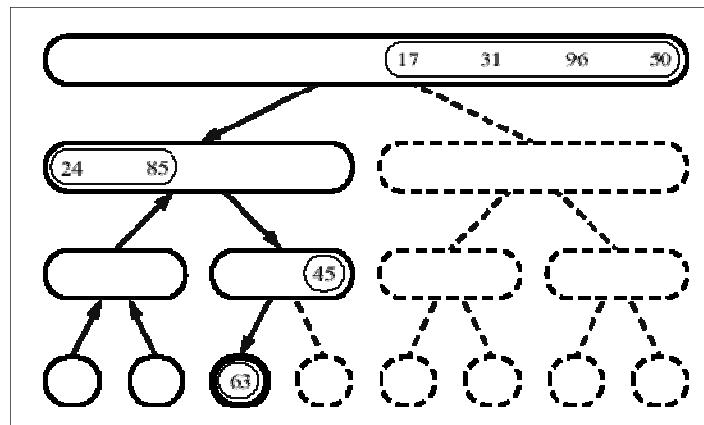
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

19

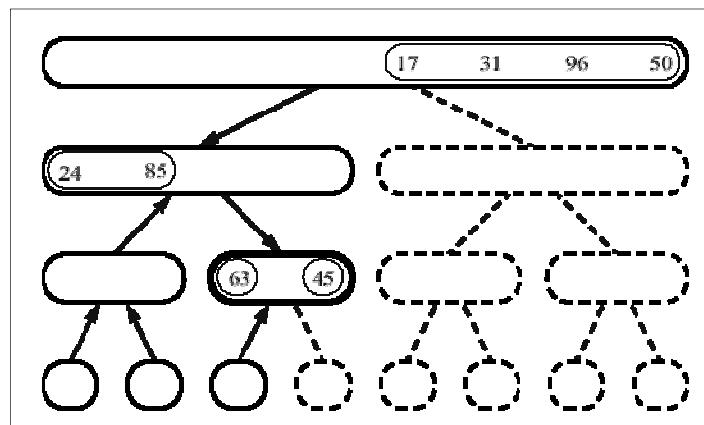
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

20

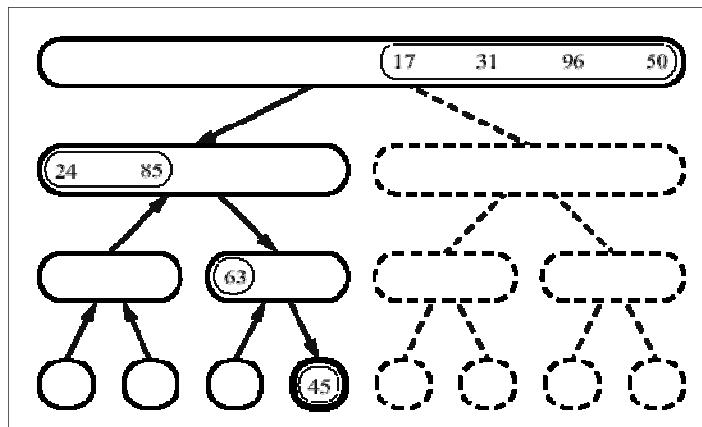
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

21

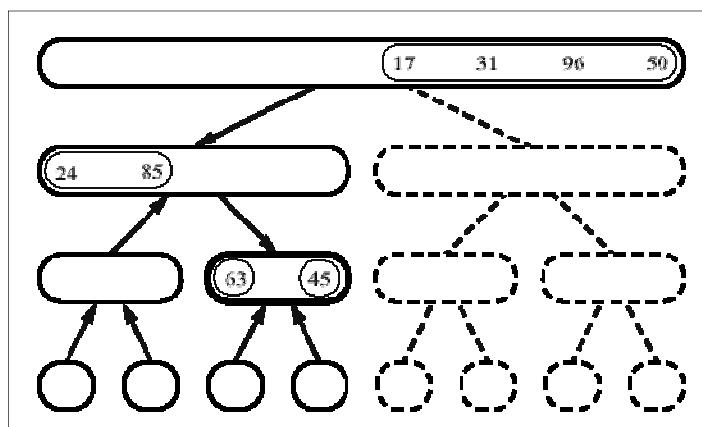
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

22

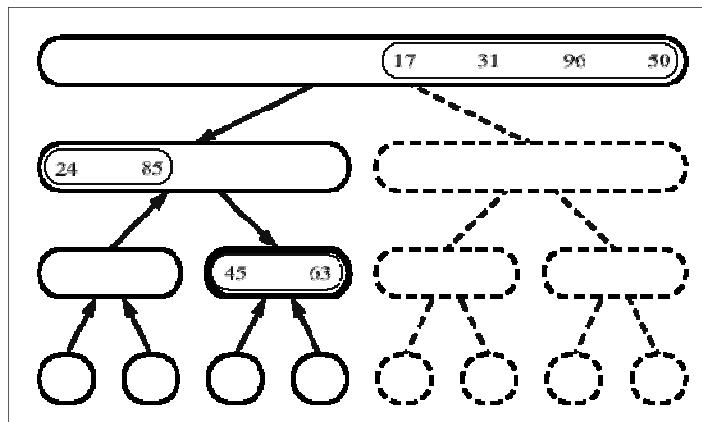
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

23

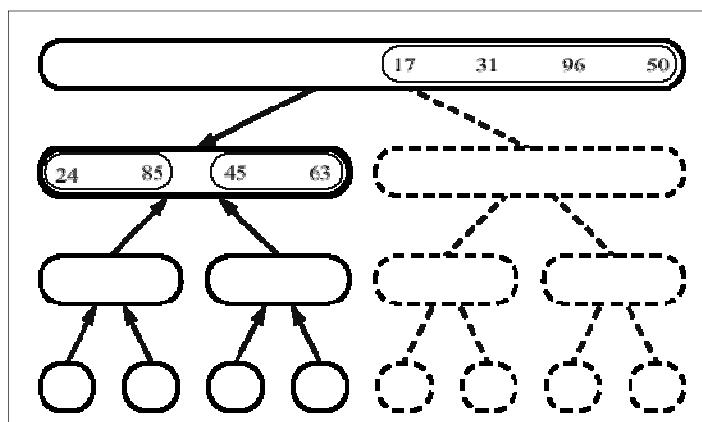
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

24

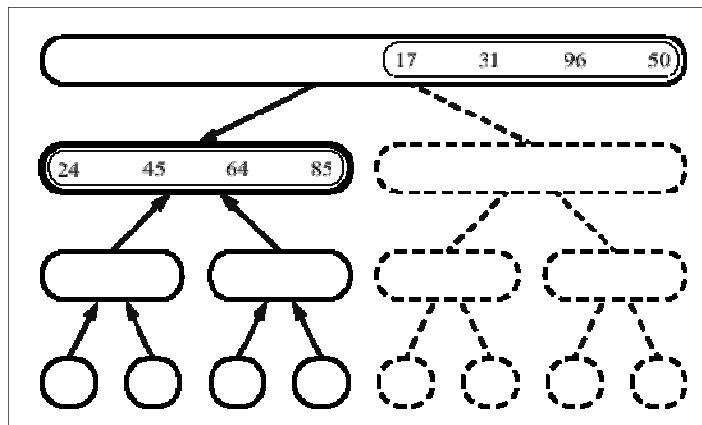
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

25

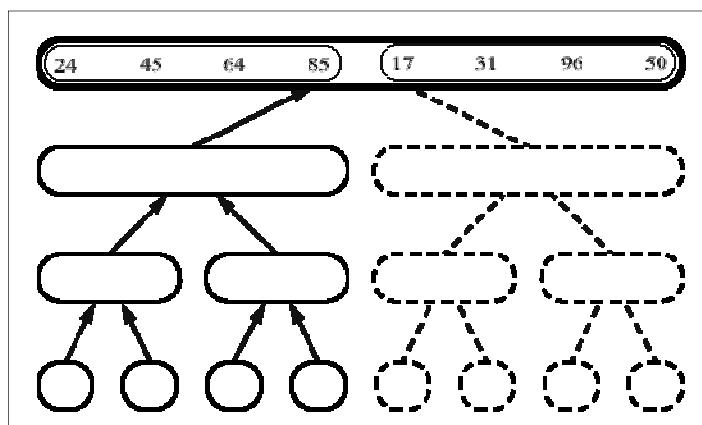
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

26

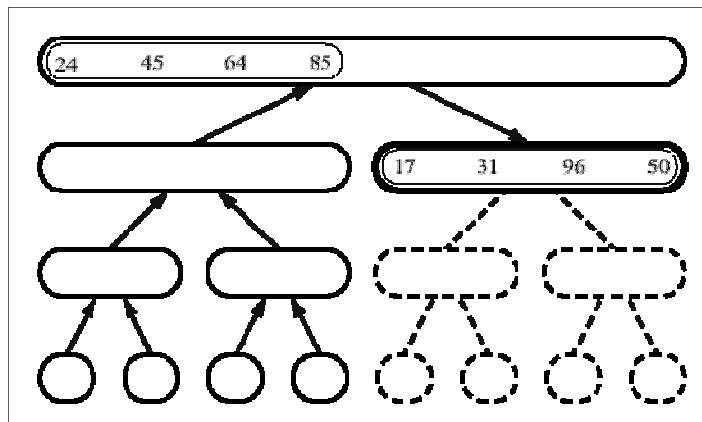
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

27

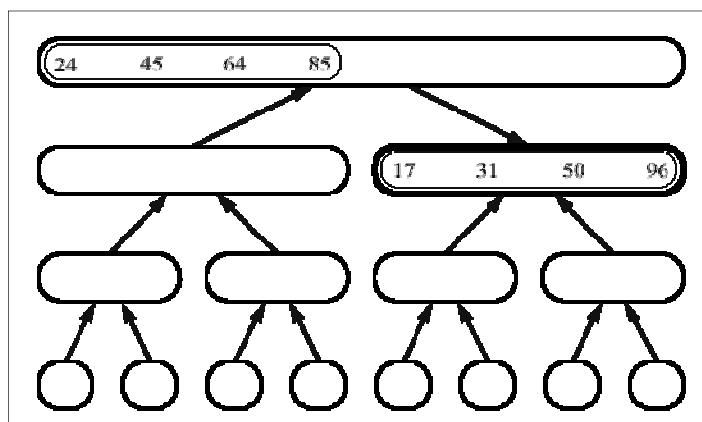
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

28

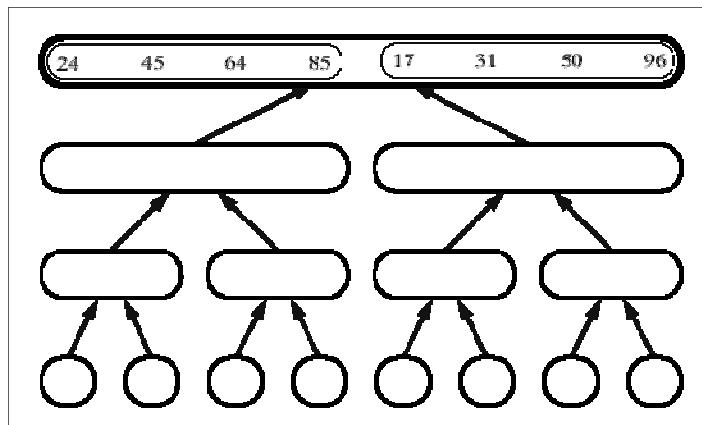
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

29

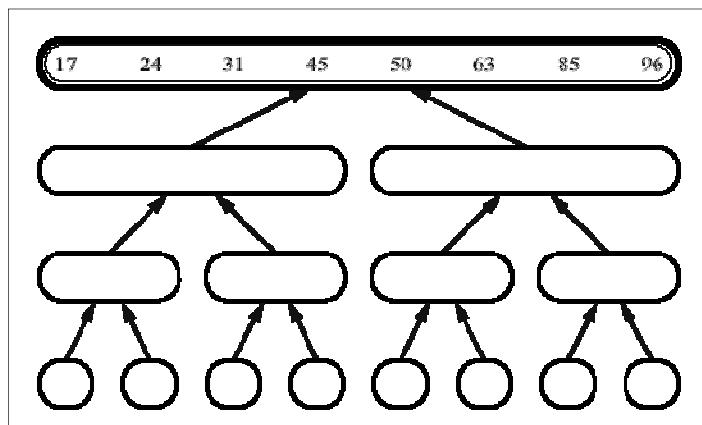
Exemplo: *Mergesort*



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

30

Exemplo: *Mergesort*

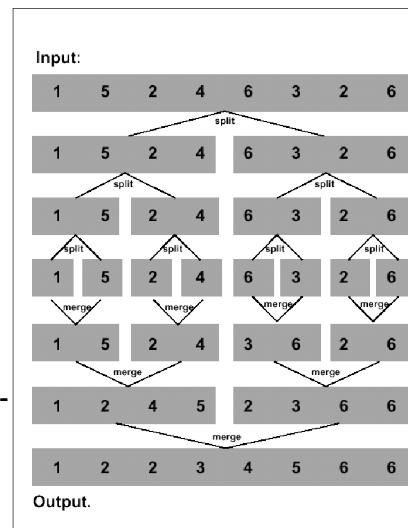


Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

31

Exemplo: *Mergesort*

- ◆ P/ ordenar n elementos:
 - Se $n = 1$, está feito!
 - Recursivamente ordenar 2 listas de $\lfloor n/2 \rfloor$ elementos
 - Combinar as duas listas ordenadas em tempo $\Theta(n)$
- ◆ Estratégia:
 - Dividir o problema em sub-problemas menores
 - Resolver recursivamente
 - Combinar as subsoluções



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

32

Exemplo: pesquisa binária

- ◆ Seja S uma sequência ordenada de n elementos, e s_x um elemento que se pretende procurar dentro de S .
- ◆ Se s_x é o elemento médio, então retorna-se s_x !
- ◆ Senão:
 - Dividir: divide-se S em duas sequências, S_1 e S_2 , com $\sim n/2$ elementos; se $s_x <$ que o elemento médio, escolhe-se S_1 para continuar; se $s_x >$ que o elemento médio, escolhe-se S_2 para continuar.
 - Conquistar: tenta-se resolver a subsequência para determinar se s_x está presente.
 - Obtém-se a solução para a sequência a partir da solução do problema para as subsequências!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

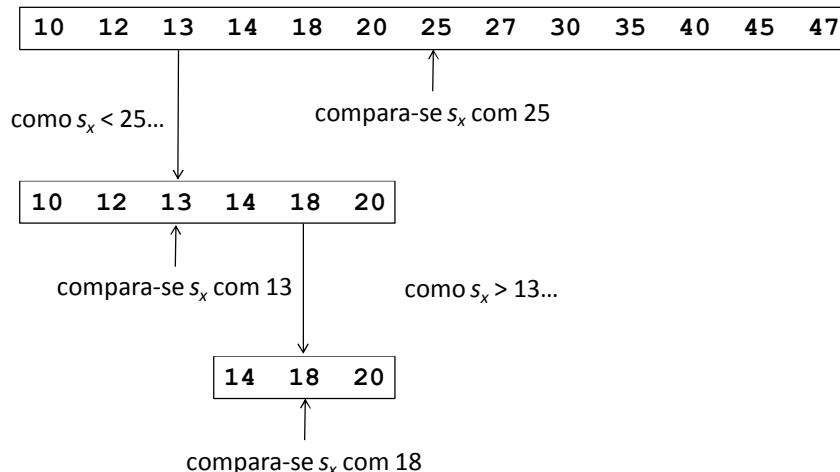
33

Exemplo: pesquisa binária

- ◆ Suponha que $s_x = 18$ e que S é dado por:
 $\{10 \ 12 \ 13 \ 14 \ 18 \ 20 \ 25 \ 27 \ 30 \ 35 \ 40 \ 45 \ 47\}$
- ↑
termo médio
- ◆ Dividir a sequência: sendo $s_x < 25$, pesquisa-se em
 $\{10 \ 12 \ 13 \ 14 \ 18 \ 20\}$
- ◆ Conquistar a subsequência, determinando-se se s_x está presente.
- ◆ Obtém-se a solução para a sequência S , pela solução da pesquisa nas subsequências. R: Sim! $s_x \in S$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

34



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

35

Problema: determinar se x está presente numa sequência ordenada S , de tamanho n .

Dados: inteiro positivo n , array ordenado S , indexado de $1..n$, uma chave x .

Resultado: $location$, localização de x em S (0 se x não está em S)

```

index location(index low, index high)
    index mid;
    if ( low > high )
        return 0;
    else
        mid = [ (low + high) / 2 ];
        if ( x == S[mid] )
            return mid;
        else if ( x < S[mid] )
            return location(low, mid - 1);
        else
            return location(mid + 1, high);

```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

36

Referências

- ◆ Mark Allen Weiss. Data Structures & Algorithm Analysis in Java. Addison-Wesley, 1999
- ◆ Steven S. Skiena. The Algorithm Design Manual. Springer 1998
- ◆ Robert Sedgewick. Algorithms in C++. Addison-Wesley, 1992

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

37

Em suma...

- ◆ Programação dinâmica (*dynamic programming*)
 - Contexto: Problemas de solução recursiva.
 - Objectivo: Minimizar tempo e espaço.
 - Forma: Induzir uma progressão iterativa de transformações sucessivas de um espaço linear de soluções.
- ◆ Algoritmos gananciosos (*greedy algorithms*)
 - Contexto: Problemas de optimização (max. ou min.)
 - Objectivo: Atingir a solução óptima, ou uma boa aproximação.
 - Forma: tomar uma decisão óptima localmente, i.e., que maximiza o ganho (ou minimiza o custo) imediato

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

38

Em suma...

- ◆ Algoritmos de retrocesso (*backtracking*)
 - Contexto: problemas sem algoritmos eficientes (convergentes) para chegar à solução.
 - Objectivo: Convergir para uma solução.
 - Forma: tentativa-erro. Gerar estados possíveis e verificar todos até encontrar solução, retrocedendo sempre que se chegar a um beco sem saída.
- ◆ Divisão e conquista (*divide and conquer*)
 - Contexto: Problemas passíveis de se conseguirem sub-dividir.
 - Objectivo: melhorar eficiencia temporal.
 - Forma: agregação linear da resolução de sub-problemas de dimensão semelhantes até chegar ao caso-base.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP, Fev. de 2011

Algoritmos

Análise de Complexidade de Algoritmos



MIEIC - 2011

Introdução

É Algoritmo: conjunto claramente especificado de instruções a seguir para resolver um problema

É Análise de algoritmos:

ó provar que um algoritmo está correcto

ó determinar recursos exigidos por um algoritmo (tempo, espaço)

É comparar os recursos exigidos por diferentes algoritmos que resolvem o mesmo problema (um algoritmo mais eficiente exige menos recursos para resolver o mesmo problema)

É prever o crescimento dos recursos exigidos por um algoritmo à medida que o tamanho dos dados de entrada cresce



2

Complexidade espacial e temporal

É Complexidade espacial de um programa ou algoritmo: espaço de memória que necessita para executar até ao fim

$S(n)$ - espaço de memória exigido em função do tamanho (n) da entrada

É Complexidade temporal de um programa ou algoritmo: tempo que demora a executar (tempo de execução)

$T(n)$ - tempo de execução em função do tamanho (n) da entrada

É Complexidade \uparrow versus Eficiência \downarrow

É Por vezes estima-se a complexidade para o "melhor caso" (pouco útil), o "pior caso" (mais útil) e o "caso médio" (igualmente útil)



3

Crescimento de funções

É Na prática, é difícil (senão impossível) prever com rigor o tempo de execução de um algoritmo ou programa

ó Obter o tempo a menos de:

É constantes multiplicativas (normalmente estas constantes são tempos de execução de operações atómicas)

É parcelas menos significativas para valores grandes de n

É Comparar crescimento

ó Comparação de funções em pontos particulares: muito dependente dos coeficientes

ó Comparação relevante: taxas de crescimento

É Avaliar taxa de crescimento

ó Em função com vários termos, crescimento é determinado pelo termo de crescimento mais rápido

ó Coeficientes constantes influenciam o andamento inicial



4

Crescimento de função composta

É Definição:

$$T(n) = O(f(n)) \quad (\text{ler: } T(n) \text{ é de ordem } f(n))$$

se e só se existem constantes positivas c e n_0 tal que

$$T(n) \leq cf(n) \text{ para todo o } n > n_0$$

É Exemplos:

ó $c_k n^k + c_{k-1} n^{k-1} + \dots + c_0 = O(n^k)$ (c_i - constantes)

ó $\log_2 n = O(\log n)$

(não se indica a base porque mudar de base é multiplicar por constante)

ó $4 = O(1)$ (usa-se 1 para ordem constante)



5

Notação $O(\bullet)$

É Notação para o crescimento relativo de funções

ó $T(n) = O(f(n))$

se existem constantes c e n' tais que $T(n) \leq c f(n)$ para $n \geq n'$

ó $T(n) = \Omega(f(n))$

se existem constantes c e n' tais que $T(n) \geq c f(n)$ para $n \geq n'$

ó $T(n) = \Theta(f(n))$

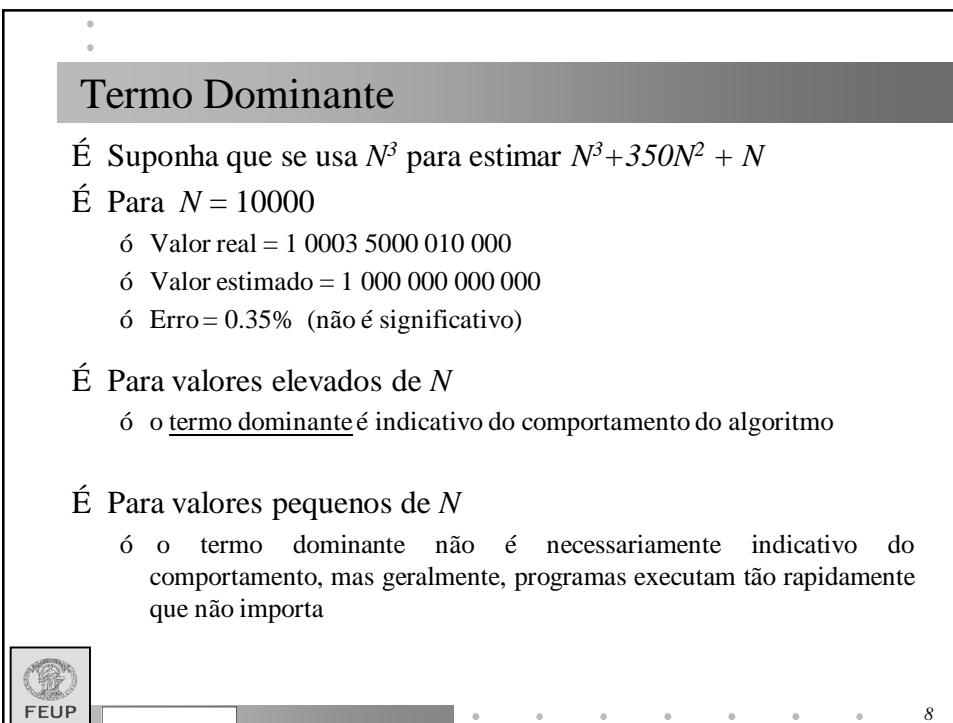
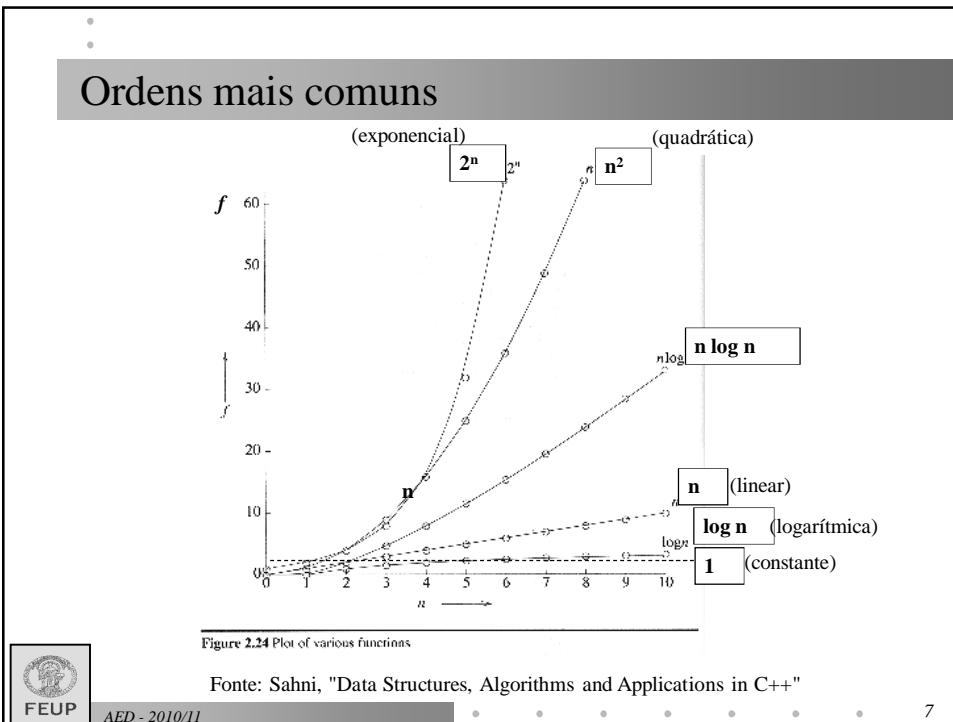
se e só se $T(n) = O(f(n))$ e $T(n) = \Omega(f(n))$

ó $T(n) = o(f(n))$

se $T(n) = O(f(n))$ e $T(n) \neq \Theta(f(n))$



6



•
•

Exemplo 1: subsequência máxima

É Problema:

ó Dado um conjunto de valores (positivos e/ou negativos) A_1, A_2, \dots, A_n
determinar a subsequência de maior soma

É A subsequência de maior soma é zero se todos os valores são negativos

É Exemplos:

-2, 11, -4, 13, -4, 2

1, -3, 4, -2, -1, 6



9

•
•

Exemplo 1 ó algoritmo 1

```
template <class Comparable>
Comparable maxSubSum1(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
        for (int j = i; j < a.size(); j++)
        {
            Comparable thisSum = 0;
            for (int k = i; k <= j; k++)
                thisSum += a[k];
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    return maxSum;
}
```



10

Exemplo 1 ó algoritmo 1

É Análise

- ó ciclo de N iterações no interior de um outro ciclo de N iterações no interior de um outro ciclo de N iterações $\Rightarrow O(N^3)$, algoritmo cúbico
- ó Valor estimado por excesso, pois alguns ciclos possuem menos de N iterações

É Como melhorar

- ó Remover um ciclo
- ó Ciclo mais interior não é necessário
- ó *thisSum* para próximo j pode ser calculado facilmente a partir do antigo valor de *thisSum*



11

Exemplo 1 ó algoritmo 2

```
template <class Comparable>
Comparable maxSubSum2(const vector<Comparable> &a)
{
    Comparable maxSum = 0;
    for (int i = 0 ; i < a.size() ; i++)
    {
        Comparable thisSum = 0;
        for (int j = i; j < a.size(); j++)
        {
            thisSum += a[j];
            if (thisSum > maxSum)
                maxSum = thisSum;
        }
    }
    return maxSum;
}
```



12

•
•
Exemplo 1 ó algoritmo 2

É Análise

- ó ciclo de N iterações no interior de um outro ciclo de N iterações $\Rightarrow O(N^2)$, algoritmo quadrático

É É possível melhorar?

- ó Algoritmo linear é melhor : tempo de execução é proporcional a tamanho de entrada (difícil fazer melhor)
- ó Se A_{ij} é uma subsequência com custo negativo, A_{iq} com $q > j$ não é a subsequência máxima



13

•
•
Exemplo 1 ó algoritmo 3

```
template <class Comparable>
Comparable maxSubSum3(const vector<Comparable> &a)
{
    Comparable thisSum = 0; Comparable maxSum = 0;
    for (int i = 0, j=0 ; j < a.size() ; j++)
    {
        thisSum += a[j];
        if (thisSum > maxSum)
            maxSum = thisSum;
        else if (thisSum < 0)
        {
            i = j+1;
            thisSum = 0;
        }
    }
    return maxSum;
}
```



14

Exemplo 1 ó algoritmo 4

- É Método õdivisão e conquistaõ
- ó Divide a sequênciæ ao meio
- ó A subseqüênciæ máximæ estã:

 - a) na primeiræ metade
 - b) na segundæ metade
 - c) começã na 1^a metade, vai até ao último elemento da 1^a metade, continua no primeiræ elemento da 2^a metade, e termina em um elemento da 2^a metade.

- ó Calcula as tréns hipóteses e determina o máximæ
- ó a) e b) calculados recursivamente
- ó c) realizado em dois ciclos:
 - É percorrer a 1^a metade da direita para a esquerda, começando no último elemento
 - É percorrer a 2^a metade da esquerda par a direita, começando no primeiræ elemento



15

Exemplo 1 ó algoritmo 4

```
template <class Comparable>
Comparable maxSubSum(const vector<Comparable> &a, int left, int
right)
{
    Comparable maxLeftBorderSum = 0, maxRightBorderSum = 0
    Comparable leftBorderSum = 0, rightBorderSum = 0;
    int center = (left + right) / 2;

    if (left == right)
        return ( a[left] > 0 ? a[left] : 0 )

    Comparable maxLeftSum = maxSubSum (a, left, center);
    Comparable maxRightSum = maxSubSum (a, center + 1, right);
```



16

•
•
•

Exemplo 1 ó algoritmo 4

```

for (int i = center ; i >= left ; i--)
{
    leftBorderSum += a[i];
    if (leftBorderSum > maxLeftBorderSum)
        maxLeftBorderSum = leftBorderSum;
}
for (int j = center +1 ; j <= right ; j++)
{
    rightBorderSum += a[j];
    if (rightBorderSum > maxRightBorderSum)
        maxRightBorderSum = rightBorderSum;
}

return max3( maxleftSum, maxRightSum,
            maxLeftBorderSum + maxRightBorderSum);
}

```



17

•
•
•

Exemplo 1 ó algoritmo 4

É Análise

- ó Seja $T(N)$ = tempo execução para problema tamanho N

- ó $T(1) = 1$ (recorda-se que constantes não interessam)

- ó $T(N) = 2 * T(N/2) + N$

É duas chamadas recursivas, cada uma de tamanho $N/2$. O tempo de execução de cada chamada recursiva é $T(N/2)$

É tempo de execução de caso c) é N



18

Exemplo 1 ó algoritmo 4

É Análise

$$\left\{ \begin{array}{l} T(N) = 2 * T(N/2) + N \\ T(1) = 1 \end{array} \right.$$

$$T(N/2) = 2 * T(N/4) + N/2$$

$$T(N/4) = 2 * T(N/8) + N/4$$

...

$$T(N) = 2 * 2 * T(N/4) + 2 * N/2 + N$$

$$T(N) = 2 * 2 * 2 * T(N/8) + 2 * 2 * N/4 + 2 * N/2 + N$$

$$T(N) = 2^k * T(N/2^k) + kN$$

$$T(1) = 1 : N/2^k = 1 \Rightarrow k = \log_2 N$$

$$T(N) = N * 1 + N * \log_2 N = O(N * \log N)$$



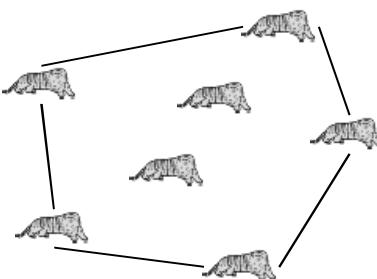
19

Exemplo 2

É Problema òbarricada dos tigresö

- ó Dados n tigres posicionados num campo, determinar o menor polígono que os cerca

nota: um segmento faz parte da cerca se todos os restantes pontos estão no mesmo lado desse segmento



Algoritmo 1

- ó Testar para cada um dos n^2 segmentos se os restantes $n-2$ pontos estão do mesmo lado.
- ó Complexidade temporal: $O(n^3)$



20

Exemplo 2

Algoritmo 2

1. Procurar o ponto mais baixo $\Rightarrow O(n)$
2. Ordenar os restantes pontos por ordem crescente de ângulos $\Rightarrow O(n \log n)$
3. Considerar que o segmento $P_1 P_2$ faz parte da cerca $\Rightarrow O(1)$
4. Para cada um dos restantes pontos P_i (*): $\Rightarrow O(n)$
 - i. Adicionar P_i à cerca
 - ii. Verificar todos os pontos P_j anteriores da cerca (por ordem inversa) e eliminá-los caso P_1 e P_i estejam em lados diferentes do segmento com os pontos P_j e P_{j-1} . Parar este processo quando um P_j não for eliminado.

(*): à medida que são eliminados não voltam a ser comparados

Complexidade temporal: $O(n \log n)$



Grafos

1

Algoritmos em Grafos: Introdução

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Março de 2011

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

2

Índice

- ◆ Revisão de conceitos e definições
- ◆ Exemplificar aplicações
- ◆ Representação
- ◆ Pesquisa em profundidade e em largura
- ◆ Ordenação topológica

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

3

Revisão de conceitos e definições

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

4

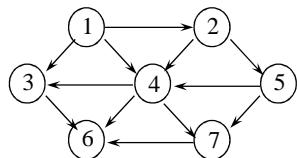
Conceito de grafo

- ◆ Grafo $G = (V, E)$
 - V – conjunto de vértices (ou nós)
 - E – conjunto de arestas (ou arcos)
 - cada aresta é um par de vértices (v, w) , em que $v, w \in V$
 - se o par for ordenado, o grafo é dirigido, ou digrafo
 - um vértice w é adjacente a um vértice v se e só se $(v, w) \in E$
 - num grafo não dirigido com aresta (v, w) e, logo, (w, v) , w é adjacente a v e v adjacente a w
 - as arestas têm por vezes associado um custo ou peso

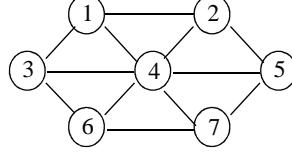
Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

5

Grafos dirigidos e não dirigidos



G1 = (Cruzamentos, Ruas)



G2 = (Cidades, Estradas)

- ◆ Algumas aplicações

- Tráfego, transporte (controlo, gestão)
- Navegação GPS
- Abastecimento de água e redes de saneamento (gestão de carga)
- Gestão de redes de energia
- Workflows e cadeias de decisão
- Planeamento e gestão de projectos
- Compiladores, sistemas de ficheiros, jogos, criptografia, redes, Internet
- Redes Bayesianas e probabilísticas (Processo de Manchester)

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

6

Caminhos e ciclos

- ◆ Caminho - sequência de vértices v_1, \dots, v_n tais que $(v_i, v_{i+1}) \in E, 1 \leq i < n$
 - comprimento do caminho é o número de arestas, $n-1$
 - se $n = 1$, caminho reduz-se a um vértice e tem comprimento 0
 - anel - caminho $v, v \Rightarrow (v, v) \in E$, comprimento 1; raro
 - caminho simples - todos os vértices distintos excepto possivelmente o primeiro e o último
- ◆ Ciclo - caminho de comprimento ≥ 1 com $v_1 = v_n$
 - num grafo não dirigido requer-se que as arestas sejam diferentes
 - DAG (Grafo Dirigido Acíclico) é um grafo dirigido sem ciclo. Para qualquer vértice v , não há nenhuma ligação dirigida começando e acabando em v .

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

7

Conectividade e densidade

◆ Conectividade:

- Grafo não dirigido é conexo sse houver um caminho a ligar qualquer par de vértices
- Digrafo com a mesma propriedade: fortemente conexo, se p/ todo $v, w \in V$ existir em G um caminho de v para w e w para v .
- Digrafo fracamente conexo (ou não fortemente conexo): se, para toda bipartição (X, Y) de seu conjunto de vértices V , alguma aresta (x, y) tem uma ponta em X e outra em Y . Ou seja, não há bipartição vazia!

◆ Densidade:

- Grafo denso – $|E| = \Theta(V^2)$
- Grafo completo – existe uma aresta entre qualquer par de nós
- Grafo esparsa – $|E| = \Theta(V)$

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

8

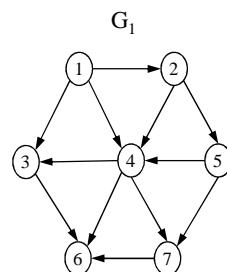
Representação

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

9

Matriz de adjacências

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



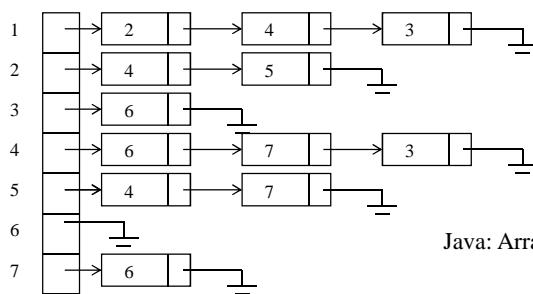
- $a[u][v] = 1$ sse $(u, v) \in E$
- elementos da matriz podem ser os pesos (0 - não há aresta)
- grafo não dirigido - matriz simétrica
- apropriada para grafos densos
- 3000 cruzamentos x 12 000 troços de ruas (4 por cruzamento)
= 9 000 000 de elementos na matriz!

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

10

Lista de adjacências

- ◆ Estrutura típica para grafos esparsos
 - para cada vértice, mantém-se a lista dos vértices adjacentes
 - vector de cabeças de lista, indexado pelos vértices
 - espaço é $O(|E| + |V|)$
 - pesquisa de adjacentes em tempo proporcional ao número destes
- ◆ Grafo não dirigido: lista com dobro do espaço

Java: `ArrayList<LinkedList<Integer>>`

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

11

Representação

- ◆ Normalmente precisamos de guardar informação adicional em cada vértice e em cada aresta (nome, peso, etc.), pelo que se opta por uma representação mais complexa, como por exemplo:

```
class Graph {
    ArrayList<Vertex> vertexSet;
}

class Vertex {
    String name;
    LinkedList<Edge> adj; //arestas a sair deste
    vértice
}

class Edge {
    Vertex dest;
    double weight;
}
```

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

12

Referências e mais informação

- ◆ “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- ◆ “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

1

Algoritmos em Grafos: Pesquisa e Ordenação Topológica

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Março de 2011

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

2

Pesquisa em profundidade e em largura

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

3

Pesquisa profundidade (depth-first search)

- ◆ Arestas são exploradas a partir vértice v mais recentemente descoberto que ainda tenha arestas a sair dele.
- ◆ Quando todas as arestas de v forem exploradas, retorna para explorar arestas que saíram do vértice a partir do qual v foi descoberto.
- ◆ Se se mantiverem vértices por descobrir, um deles é seleccionado como a nova fonte e o processo de pesquisa continua a partir daí.
- ◆ Todo o processo é repetido até todos os vértices serem descobertos.

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

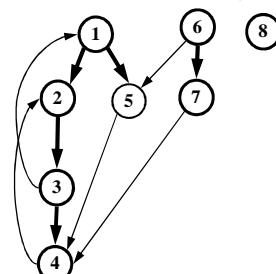
4

Pesquisa em profundidade

```

1: class Graph { ...
2: void dfs() {
3:   for (Vertex v : vertexSet)
4:     v.visited = false;
5:   for (Vertex v : vertexSet)
6:     if (! v.visited)
7:       dfs(v);
        //v passa a ser a raiz de uma
        //árvore na floresta dfs.
8: }
9: void dfs( Vertex v ) {
10: v.visited = true;
11:   // fazer qualquer coisa c/ v aqui
12:   for(Edge e : v.adj)
13:     if( ! e.dest.visited )
14:       dfs( e.dest );
15:     // ou aqui
16:   }
17: }
```

Exemplo em grafo dirigido (vértices numerados por ordem de visita e dispostos por profundidade de recursão):



Arestas a traço forte: árvore de expansão em profundidade

Na DFS podem ser produzidas várias árvores, porque a pesquisa pode ser repetida a partir de várias fontes (ao contrário da BFS que só produz uma). O conjunto das várias árvores é conhecido como Floresta DFS.

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

5

Pesquisa largura (breadth-first search)

- ◆ Dado um vértice s , explora-se sistematicamente o grafo descobrindo todos os vértices a que se pode chegar a partir de s . Só depois é que passa para outro vértice.
- ◆ Calcula a distância (número mais pequeno de arestas) de s para qualquer outro vértice.
- ◆ Produz uma árvore BFS com raiz s . Para qualquer vértice v a que se possa chegar a partir de s , o caminho na árvore BFS é o mais curto no grafo.

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

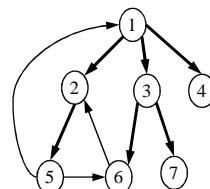
6

Pesquisa em largura

```

1: void bfs( Vertex v ) {
2:     Queue<Vertex> q = new Queue<Vertex>();
3:     q.add(v); // insere root fila (FIFO)
4:     v.discovered = true;
5:     while (q.size() > 0) {
6:         Vertex v = q.remove();
// Procura vértices adjacentes ao obtido na
// linha 6 e adiciona-os à fila
7:         for(Edge e : v.adj) {
8:             Vertex w = e.dest;
9:             if( ! w.discovered) {
10:                 w.discovered = true;
11:                 q.add(w);
12:             }
13:         }
// marca vértice v como explorado
14:         v.explored = true;
15:     }
16: }
```

Exemplo em grafo dirigido
(vértices numerados por ordem de visita e dispostos por níveis de distância ao vértice inicial):



Arestas a traço forte: árvore de expansão em largura

BFS é um dos métodos mais simples e é o arquétipo para muitos algoritmos importantes de grafos. Exemplo: Prim's Minimum-Spanning Tree e Dijkstra Single-Source Shortest-paths.

Funciona em grafos dirigidos e não dirigidos.

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

7

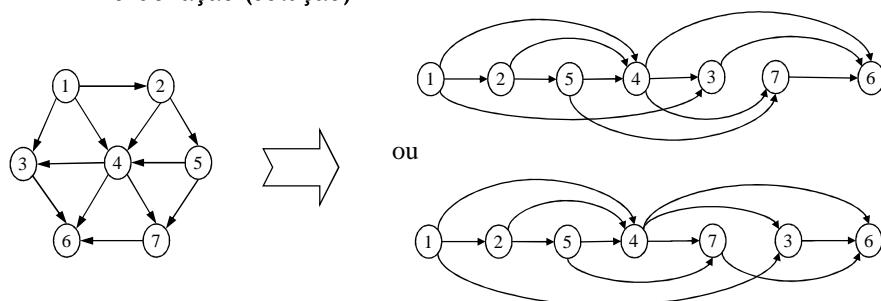
Ordenação topológica

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

8

Problema

- ◆ Ordenação linear dos vértices de um DAG (Grafo Acíclico Dirigido) tal que, se existe uma aresta (v, w) no grafo, então v aparece antes de w
 - Impossível se o grafo for cíclico
 - Não é necessariamente única. Pode existir mais do que uma ordenação (solução)

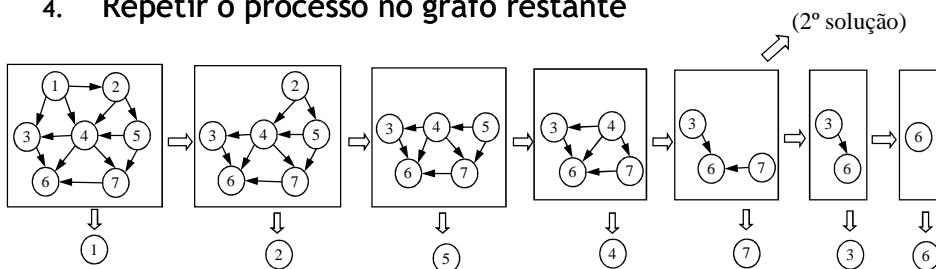


Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

9

Método básico

1. Descobrir um vértice sem arestas de chegada (indegree=0)
2. Imprimir o vértice
3. Eliminá-lo e às arestas que dele saem
4. Repetir o processo no grafo restante



Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

10

Algoritmo de ordenação topológica

- ◆ Refinamento do método básico:
 - simular eliminação actualizando indegree dos vértices adjacentes
 - memorizar numa estrutura auxiliar vértices por imprimir c/ indegree=0
- ◆ Dados de entrada:
 - V - conjunto de vértices
 - adj(v) - conjunto (ou lista) de vértices adjacentes a cada vértice v
 - ou conj. de arestas que saem de v, que por sua vez indicam vértices adj.
- ◆ Dados de saída
 - T - sequência (ou lista) dos vértices por ordem topológica
 - ou numTop(v) - número atribuído a cada vértice v por ordem topológica
- ◆ Dados temporários
 - indegree(v) - nº de arestas que chegam a v, partindo de vértices por visitar
 - C - conjunto de vértices por visitar cujo indegree é 0 (candidatos)

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

11

Algoritmo de ordenação topológica

◆ Método:

1. for each $v \in V$ do $\text{indegree}(v) \leftarrow 0$
2. for each $v \in V$ do for each $w \in \text{adj}(v)$ do $\text{indegree}(w) \leftarrow \text{indegree}(w) + 1$
3. $C \leftarrow \{\}$ // Pode ser uma fila (queue)
4. for each $v \in V$ do if $\text{indegree}(v) = 0$ then $C \leftarrow C \cup \{v\}$
5. $T \leftarrow []$ // Pode ser uma lista LinkedList
6. while $C \neq \{\}$ do
7. $v \leftarrow \text{remove-one}(C)$
8. $T \leftarrow T \text{ concatenado-com } [v]$
9. for each $w \in \text{adj}(v)$ do
10. $\text{indegree}(w) \leftarrow \text{indegree}(w) - 1$
11. if $\text{indegree}(w) = 0$ then $C \leftarrow C \cup \{w\}$
12. if $|T| \neq |V|$ then Fail("O grafo tem ciclos")

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

12

Análise do algoritmo

- ◆ As diferentes escolhas do próximo vértice no ponto 7 dão as diferentes soluções possíveis
- ◆ Se as inserções e eliminações em C forem efectuadas em tempo constante (usando por exemplo uma fila FIFO), algoritmo pode ser executado em tempo $O(|V|+|E|)$
 - o corpo do ciclo de actualização do indegree (passos 9, 10, 11) é executado no máximo uma vez por aresta
 - as operações de inserção e remoção na fila (nos passos 4, 7 e 11) são executadas no máximo uma vez por vértice
 - a inicialização leva um tempo proporcional ao tamanho do grafo

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

13

Implementação (1/2)

```

public LinkedList<Vertex> topsort()
    throws CycleFoundException
{
    // Inicializações
    for (Vertex v : vertexSet)
        v.indegree = 0;
    for (Vertex v : vertexSet)
        for (Edge e : v.adj)
            e.dest.indegree++;

    Queue<Vertex> C = new LinkedList<Vertex>();
    for (Vertex v : vertexSet)
        if (v.indegree == 0)
            C.add(v);

    LinkedList<Vertex> T = new LinkedList<Vertex>();

```

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

14

Implementação (2/2)

```

// Trabalho
while (!C.isEmpty()) {
    Vertex v = C.remove();
    T.add(v);
    for (Edge e : v.adj) {
        Vertex w = e.dest;
        w.indegree--;
        if (w.indegree == 0)
            C.add(w);
    }
}
// Terminação
if (T.size() != vertexSet.size())
    throw new CycleFoundException();
return T;
}

```

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

15

Aplicações

- ◆ Grafos Acíclicos Dirigidos (DAG) são usados em aplicações onde é necessário indicar a precedência entre eventos.
- ◆ Exemplo: Escalonamento de Sequências de tarefas.
- ◆ A ordenação topológica de um DAG dá-nos uma ordem (sequência) dos eventos (tarefas, trabalhos, etc.) representados nesse DAG.

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

16

Referências e mais informação

- ◆ “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- ◆ “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP, Março de 2011

1

Algoritmos em Grafos: Caminho mais curto

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Março de 2011

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

2

Índice

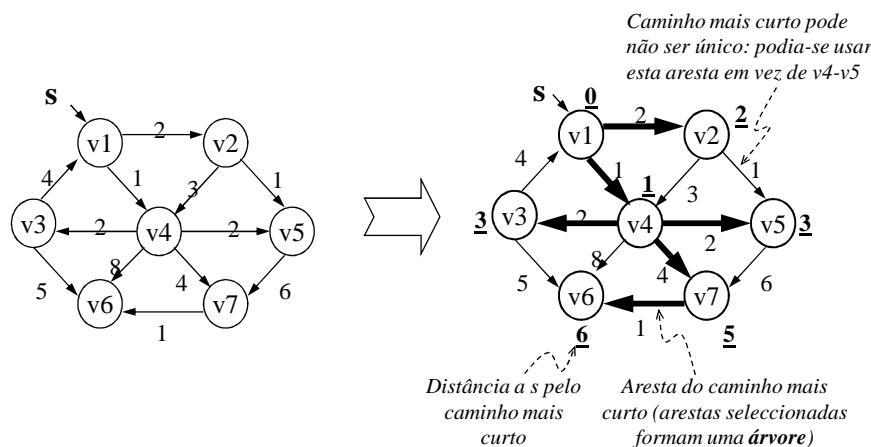
- ◆ Caminhos mais curtos dum vértice para todos os outros
 - Caso de grafos dirigidos não pesados
 - Caso de grafos dirigidos pesados
 - Caso de grafos dirigidos com arestas de peso negativo
 - Caso de grafos dirigidos acíclicos
- ◆ Caminhos mais curtos entre todos os pares de vértices
- ◆ Caminho mais curto entre dois vértices
- ◆ Optimizações para redes viárias
- ◆ Aplicação à gestão de projectos

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

3

Caminhos mais curtos dum vértice para todos os outros

Dado um grafo pesado $G = (V, E)$ e um vértice s , obter o caminho mais “curto” (de peso total mínimo) de s para cada um dos outros vértices em G



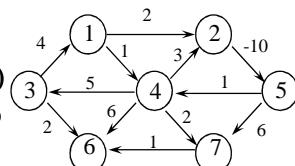
Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

4

Variantes

(Em relação ao caso base: grafo dirigido, fortemente conexo, pesos > 0)

- ◆ Grafo não dirigido
 - Mesmo que grafo dirigido com pares de arestas simétricas
- ◆ Grafo não conexo
 - Pode não existir caminho para alguns vértices, ficando distância infinita
- ◆ Grafo não pesado
 - Mesmo que peso 1 (mais curto = com menos arestas)
 - Existe um algoritmo mais eficiente para este caso do que p/ caso base
- ◆ Arestras com custos negativos
 - Ciclos com custo negativo tornam o caminho mais curto indefinido
(de v4 a v7 o custo pode ser 2 ou -1 ou -4 ou ...)
 - Exige algoritmo menos eficiente para este caso do que para o caso base



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

5

Aplicações

- ◆ Problemas de encaminhamento (*routing*)
 - Encontrar o melhor percurso numa rede viária
 - Encontrar o melhor percurso de avião
 - Encontrar o melhor percurso de metro
 - Encaminhamento de tráfego em redes informáticas
- ◆ Problemas de planeamento:
 - Por onde passar cabos nas ruas, desde uma sede até um conjunto de filiais, por forma a minimizar a distância de cada filial até à sede

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

6

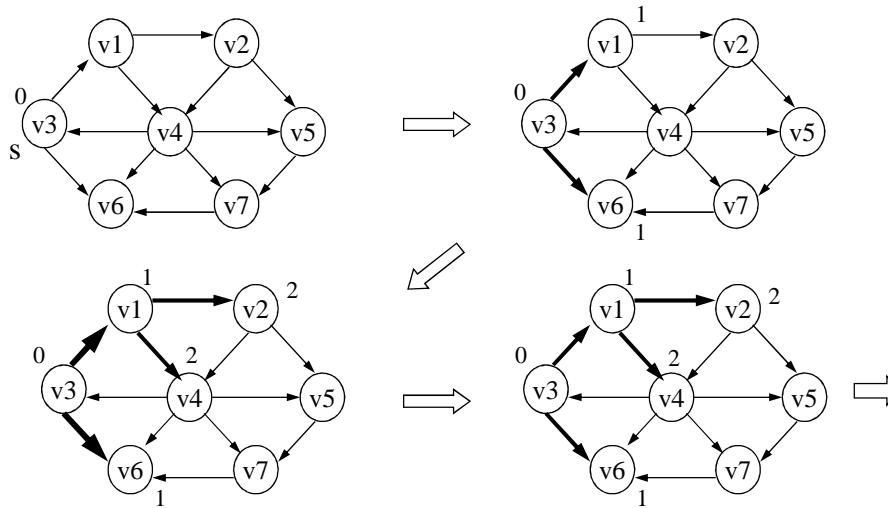
Caso de grafo dirigido não pesado

- ◆ Método básico
 1. Marcar o vértice s com distância 0 e todos os outros com distância ∞
 2. Entre os vértices já alcançados ($\text{distância} \neq \infty$) e não processados (no passo 3), escolher para processar um vértice v marcado com distância mínima
 3. Analisar os adjacentes de v , marcando os que ainda não tinham sido alcançados ($\text{distância} = \infty$) c/ a distância de v mais 1, bem como as arestas correspondentes
 4. Passar ao passo 2 se existirem mais vértices para processar
- ◆ Esta ordem de progressão por distâncias crescentes (1º vértices a distância 0, depois a distância 1, etc.) é crucial para garantir eficiência
 - Distância fica definitivamente definida ao alcançar um vértice pela 1ª vez; ao alcançar por um 2º caminho, a distância nunca é inferior
 - Este tipo de pesquisa em grafos designa-se por **pesquisa em largura** (semelhante à travessia por níveis de uma árvore)

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

7

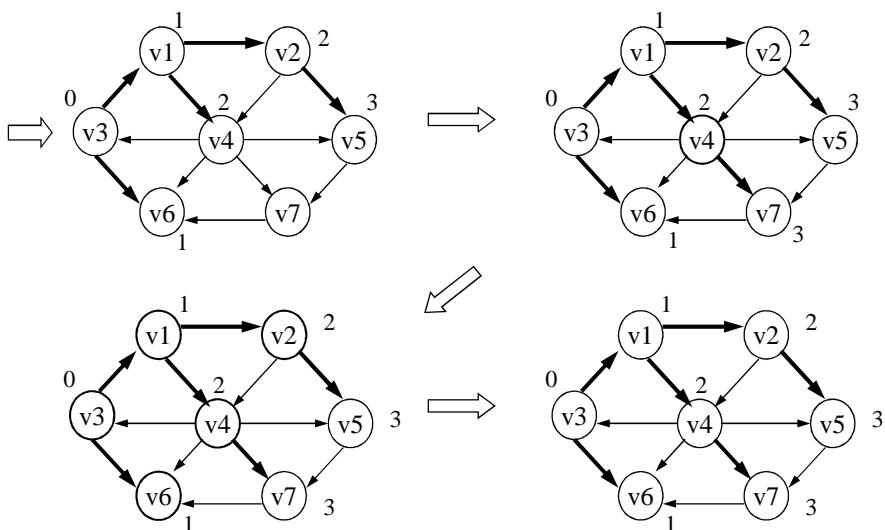
Evolução da marcação do grafo (1/2)



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

8

Evolução da marcação do grafo (2/2)



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

9

Estruturas de dados

- ◆ Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar, garante-se a ordem de progressão pretendida
- ◆ Associa-se a cada vértice a seguinte informação:
 - *dist*: distância ao vértice inicial
 - *path*: vértice antecessor no caminho mais curto (inicializado com null)

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

10

Implementação

```
void shortestPathUnweighted(Vertex s) {
    for (Vertex v : vertexSet) {
        v.path = null;
        v.dist = INFINITY;
    }
    s.dist = 0;
    Queue<Vertex> q = new LinkedList<Vertex>();
    q.add(s); // na cauda
    while( ! q.isEmpty() ) {
        v = q.remove(); // da cabeça
        for (Edge e : v.adj) {
            Vertex w = e.dest;
            if (w.dist == INFINITY) {
                w.dist = v.dist + 1;
                w.path = v;
                q.add(w);
            }
        }
    }
}
```

Tempo de execução:
 $O(|E| + |V|)$

Espaço auxiliar:
 $O(|V|)$

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

11

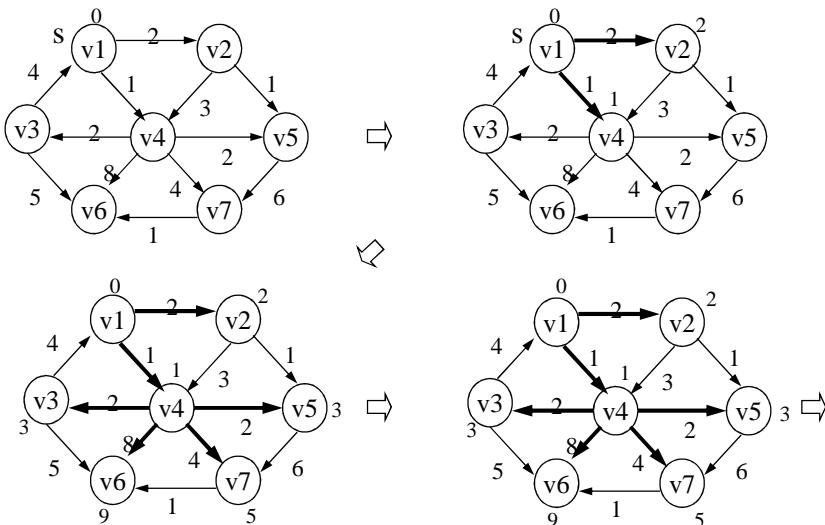
Caso de grafo dirigido pesado

- ◆ Método básico semelhante ao caso de grafo não pesado
 - Distância obtém-se somando pesos das arestas em vez de 1
 - Próximo vértice a processar continua a ser o que tem distância mínima
 - Mas já não é necessariamente o mais antigo \Rightarrow Obriga a usar fila de prioridades (com mínimo à cabeça) em vez duma fila simples
 - Esta ordem é crucial para garantir que a distância (e caminho mais curto) dos vértices já processados ao vértice inicial não é mais alterada, assumindo que não há pesos negativos
 - Mas já pode ser necessário rever em baixa a distância de um vértice alcançado e ainda não processado (vértice na fila) \Rightarrow Obriga a usar fila de prioridades alteráveis
- ◆ Restrição: só é válido se não existirem custos negativos
- ◆ É um algoritmo ganancioso: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância)

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

12

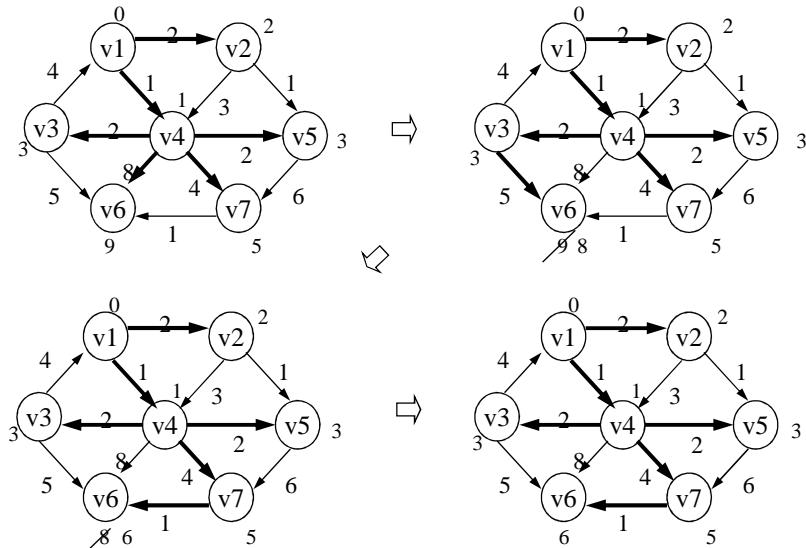
Evolução da marcação do grafo (1/2)



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

13

Evolução da marcação do grafo (2/2)



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

14

Algoritmo de Dijkstra (modificado)

```

1. void Dijkstra(Vertex s) {
2.     for (Vertex v : vertexSet) {v.path = null; v.dist = INFINITY;}
3.     s.dist = 0;
4.     PriorityQueue<Vertex> q = new PriorityQueue<Vertex>();
5.     q.insert(s);
6.     while ( ! q.isEmpty() ) {           Não serve versão da biblioteca do Java
7.         Vertex v = q.extractMin(); // remove vértice com dist mínimo
8.         for(Edge e: v.adj) {
9.             Vertex w = e.dest;
10.            if (v.dist + e.weight < w.dist) {
11.                w.dist = v.dist + e.weight;
12.                w.path = v;
13.                if (w.queueIndex == -1) // w ∉ q (ver a seguir)
14.                    q.insert(w);    Assumindo que as arestas não
15.                else               têm pesos negativos, equivale a
16.                    q.decreaseKey(w); testar que w.dist antes da
17.                }                   actualização era INFINITY
18.            }
19.        }
20.    }

```

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

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

15

Eficiência de decreaseKey

- ◆ Suponhamos que a fila de prioridades é implementada com um heap (array) com o mínimo à cabeça e seja n o tamanho do heap
- ◆ Método naive:
 - 1) Procurar sequencialmente no array objecto cuja prioridade se quer alterar - $O(n)$
 - 2) Subi-lo (ou descê-lo) na árvore até restabelecer o invariante da árvore (cada nó menor ou igual que os filhos) - $O(\log n)$
 - Total: $O(n)$ - Mau!
- ◆ Método optimizado:
 - Cada objecto colocado no heap guarda a sua posição no heap (queueIndex)
 - (Min)Heap: [v2 v3 v5 v4]
 - Objectos: v2(pri=3, idx=0), v3(pri=6, idx=1), v4(pri=7, idx=3), v5(pri=4, idx=2), v6(.., idx=-1)
 - Não é necessário o passo 1), logo o tempo total é $O(\log n)$
 - Introduz um *overhead* mínimo nas inserções e eliminações (cada vez que se insere/move um objecto no heap, queueIndex tem de ser actualizado)
 - Com Fibonacci Heaps consegue-se fazer decreaseKey em tempo amortizado $O(1)$, mas são muito mais complexos e geralmente não compensam

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

16

Eficiência do algoritmo de Dijkstra

- ◆ Tempo de execução é $O(|V| + |E| + |V| \log |V| + |E| \log |V|)$, ou simplesmente $O(|E| \log |V|)$ se $|E| > |V|$
 - $O(|V| \log |V|)$ - extração e inserção na fila de prioridades
 - O nº de extrações e inserções é $|V|$
 - Cada operação destas pode ser feita em tempo logarítmico no tamanho da fila, que no máximo é $|V|$
 - $O(|E| \log |V|)$ - decreaseKey
 - Feito no máximo $|E|$ vezes (uma vez por cada aresta)
 - Cada operação destas pode ser feita em tempo logarítmico no tamanho da fila, que no máximo é $|V|$
- ◆ Pode ser melhorado para $O(|V| \log |V|)$ com Fibonacci Heaps
 - Ver a 2ª referência; mas em geral não compensa
- ◆ Nota: O algoritmo proposto inicialmente por Dijkstra não mencionava filas de prioridades, e tinha uma eficiência $O(|V|^2)$

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

17

Caso de arestas com peso negativo

- ◆ Pode ser necessário processar cada vértice mais do que uma vez
- ◆ **Algoritmo de Bellman-Ford:** semelhante ao algoritmo de Dijkstra, mas com uma fila simples (como no caso de grafo sem pesos) em vez duma fila de prioridades (logo não precisa de decreaseKey - passos 15 e 16)
- ◆ Se não existirem ciclos com peso negativo, cada vértice é processado no máximo $|V|$ vezes e o tempo de execução no pior caso é $O(|V| \cdot |E|)$
- ◆ Se existirem ciclos com peso negativo, o problema não tem solução
- ◆ Assim que um vértice é processado mais do que $|V|$ vezes, podemos concluir que existem ciclos com peso negativo e parar o algoritmo

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

18

Caso de grafos acíclicos

- ◆ Simplificação do algoritmo de Dijkstra
 - Processam-se os vértices por ordem topológica
 - Suficiente para garantir que um vértice processado jamais pode vir a ser alterado, pois não há arestas a entrar
 - Combina-se a ordenação topológica com a actualização das distâncias e caminhos numa só passagem
 - Basta usar a fila simples da ordenação topológica, não é necessário usar uma fila de prioridades
 - Tempo de execução $O(|V| + |E|)$
- ◆ Aplicações
 - Processos irreversíveis
 - não se pode regressar a um estado passado (certas reacções químicas)
 - deslocação entre dois pontos em esqui (sempre descendente)
 - Gestão de projectos
 - Projecto composto por actividades com precedências acíclicas (não se pode começar uma actividade sem ter acabado uma precedente) - Ver adiante

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

19

Caminho mais curto entre todos os pares de vértices

- ◆ Execução repetida do algoritmo de Dijkstra (ganancioso): $O(|V| \cdot |E| \cdot \log |V|)$
 - Bom se o grafo for esparsa ($|E| \sim |V|$)
- ◆ Algoritmo de Floyd-Warshall, baseado em programação dinâmica: $O(|V|^3)$
 - Melhor que o anterior se o grafo for denso ($|E| \sim |V|^2$)
 - Mesmo em grafos pouco densos pode ser melhor porque código é mais simples
 - Usa matriz de adjacências $W[i,j]$ com pesos (∞ quando não há aresta; 0 quando $i=j$)
 - Calcula matriz de distâncias mínimas $D[i,j]$
 - Em cada iteração k (de 0 a $|V|$), $D[i,j]$ tem a distância mínima do vértice i a j , podendo usar vértices intermédios apenas do conjunto $\{1, \dots, k\}$
 - Fórmula de recorrência: $D[i,j]^{(k)} = \min(D[i, j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)})$, $k=1\dots|V|$
Começando em: $D[i,j]^{(0)} = W[i,j]$
 - Minimizar memória: actualize-se matriz em cada iteração k , em vez de criar nova
 - Mantém-se ao mesmo tempo uma matriz $P[i,j]$ que indica o vértice anterior a j no caminho mais curto de i para j

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

20

Caminho mais curto entre dois vértices

- ◆ Problema muito importante na prática
 - Exemplo: caminho mais curto entre dois pontos num mapa de estradas
- ◆ Não se conhece algoritmo mais eficiente a resolver este problema do que a resolver o mais geral (de um vértice para todos os outros)
- ◆ Portanto, acha-se o caminho mais curto da origem para todos os outros, e selecciona-se depois o caminho da origem para o destino pretendido
- ◆ Optimização: parar assim que chega a vez de processar o vértice de destino
 - Num mapa de estradas, ajuda para distâncias curtas, mas não para distâncias longas

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

21

Optimizações para redes viárias (1)

- ◆ Tirar partido da natureza hierárquica da rede
 - Vias de trânsito local (dentro de cidade), vias de trânsito de longa distância (entre cidades), etc.
 - Pré processa-se o mapa (achando caminhos mais curtos entre todos os pares de vértices) para determinar o grau de localidade $L(x)$ de cada vértice e aresta x :

$$L(x) = \max \{ \min(\text{dist-euclidiana}(s, x), \text{dist-euclidiana}(x, t)) \mid s, t \in V \wedge x \in \text{shortest-path}(s, t) \}$$
 - Ao procurar o caminho mais curto entre dois vértices s e t , ignoram-se todos as arestas e vértices x tais que

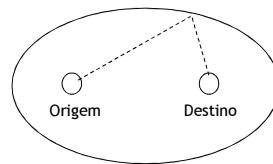
$$\min(\text{dist-euclidiana}(x, s), \text{dist-euclidiana}(x, t)) > L(x)$$

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

22

Optimizações para redes viárias (2)

- ◆ Tirar partido da natureza geométrica da rede
 - Se a rede estiver bem concebida, a distância mais curta por estrada entre dois pontos não difere da distância Euclidiana (em linha recta) a menos de um certo valor (aditivo ou multiplicativo)
 - Pré processa-se o mapa (achando caminhos mais curtos entre todos os pares de vértices) para obter este valor
 - Ao procurar o caminho mais curto entre dois pontos, usa-se o valor anterior para determinar a distância máxima por estrada, e limita-se a zona de procura a uma elipse de corda igual a essa distância máxima



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

23

Referências e mais informação

- ◆ “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- ◆ “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

1

Algoritmos em Grafos: Caminho mais curto

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

CAL, MIEIC, FEUP

Março de 2011

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

2

Aplicação à gestão de projectos

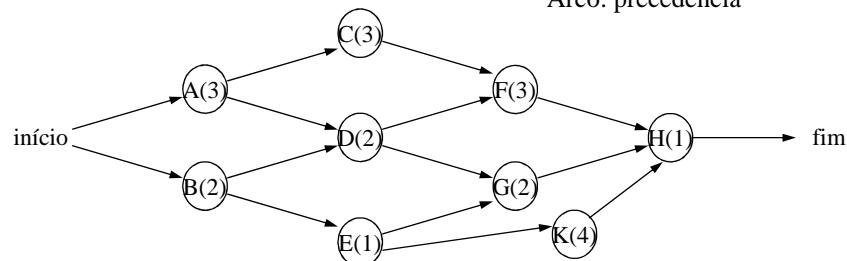
Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

3

Grafo Nó-Actividade

DAG

Nó: actividade e tempo associado
Arco: precedência



Qual a duração total mínima do projecto?

Que actividades podem ser atrasadas e por quanto tempo (sem aumentar a duração do projecto)?

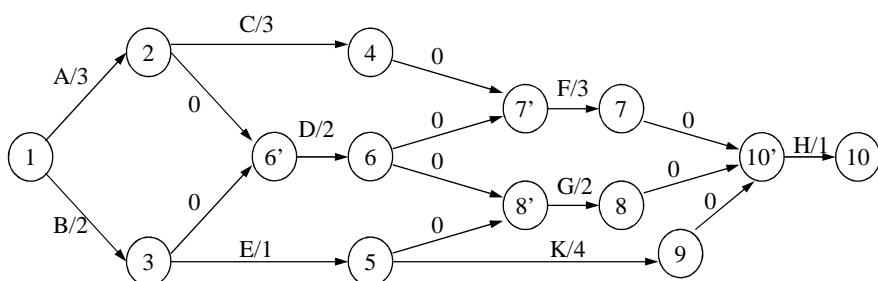
Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

4

Reformulação em Grafo Nó-Evento

DAG

Nó: evento - completar actividade
Arco: actividade



- introduzem-se nós e arcos extra para garantir precedências (só no caso de actividades com mais que uma antecessora)

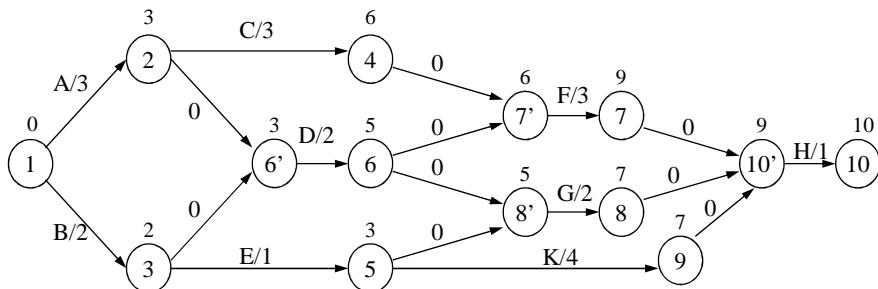
Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

5

Menor Tempo de Conclusão

- menor tempo de conclusão de uma actividade
 \Leftrightarrow caminho mais comprido do evento inicial ao nó de conclusão da actividade
- problema (se grafo não fosse acíclico): ciclos de custo positivo
- adaptar algoritmo de caminho mais curto
- $\diamond \quad MTC(1) = 0$
 $MTC(w) = \max(MTC(v) + c(v,w))$
 $(v, w) \in E$

MTC : usar ordem topológica



Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

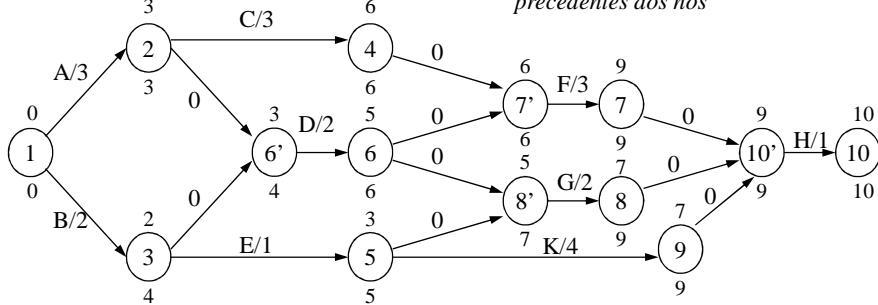
6

Último Tempo de Conclusão

- último tempo de conclusão: mais tarde que uma actividade pode terminar sem comprometer as que se lhe seguem
- $UTC(n) = MTC(n)$
 $UTC(v) = \min(UTC(w) - c(v, w))$
 $(v, w) \in E$

UTC : usar ordem topológica inversa

valores calculados em tempo linear
mantendo listas de adjacentes e de
precedentes dos nós



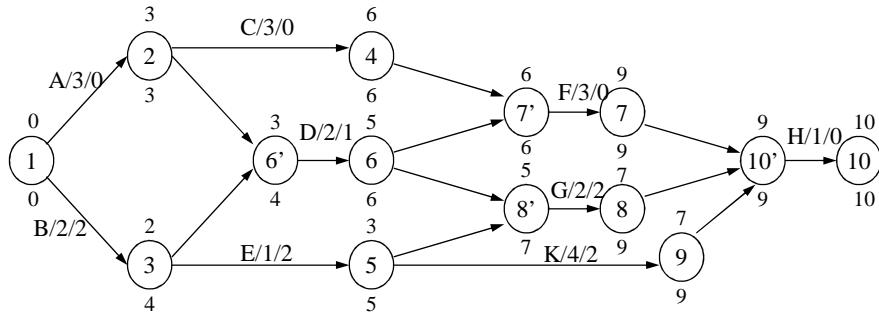
Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

7

Folgas nas actividades

- folga da actividade

$$\text{folga}(v,w) = \text{UTC}(w) - \text{MTC}(v) - c(v,w)$$



Caminho crítico: só actividades de folga nula (há pelo menos 1)

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

8

Referências e mais informação

- ◆ “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006
- ◆ “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001

Algoritmos em Grafos: Caminho mais curto • CAL - MIEIC/FEUP, Março de 2011

Algoritmos em grafos: árvore de expansão mínima (*minimum spanning tree*)

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
FEUP, MIEIC, CPAL, 2010/2011

Árvore de expansão mínima

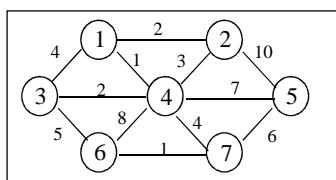
Árvore que liga todos os vértices do grafo usando arestas com um custo total mínimo

- caso do grafo não dirigido
 - grafo tem que ser conexo
 - árvore \Rightarrow grafo conexo acíclico
 - número de arestas = $|V| - 1$
- exemplo de aplicação: cablamento de uma casa
 - vértices são as tomadas
 - arestas são os comprimentos dos troços

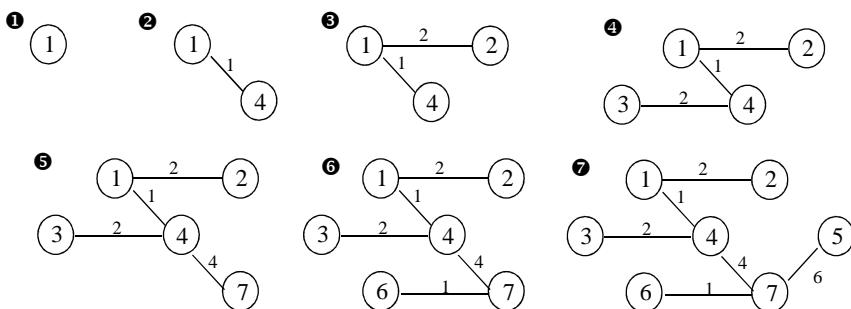
Algoritmo de Prim

- expandir a árvore por adição sucessiva de arestas e respectivos vértices
 - critério de seleção: *escolher a aresta (u,v) de menor custo tal que u já pertence à árvore e v não* (ganancioso)
 - início: um vértice qualquer
- idêntico ao algoritmo de Dijkstra para o caminho mais curto
 - informação para cada vértice
 - $\text{dist}(v)$ é o custo mínimo das arestas que ligam a um vértice já na árvore
 - $\text{path}(v)$ é o último vértice a alterar $\text{dist}(v)$
 - $\text{known}(v)$ indica se o vértice já foi processado (i.e., já pertence à árvore)
 - diferença na regra de actualização: *após a seleção do vértice v , para cada w não processado, adjacente a v , $\text{dist}(w) = \min\{\text{dist}(w), \text{cost}(v,w)\}$*
 - tempo de execução
 - $O(|V|^2)$ sem fila de prioridade
 - $O(|E| \log |V|)$ com fila de prioridade

Evolução do algoritmo de Prim



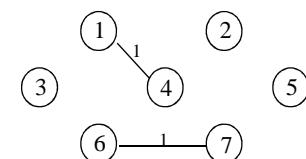
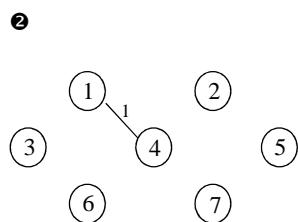
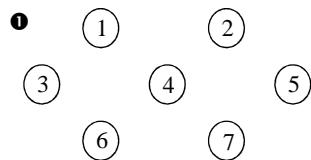
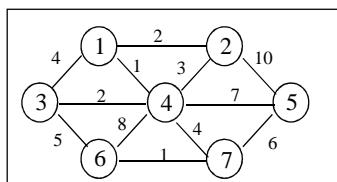
v	known	dist	path
1	1	0	0
2	1	2	1
3	1	2	4
4	1	1	1
5	1	6	7
6	1	1	7
7	1	4	4



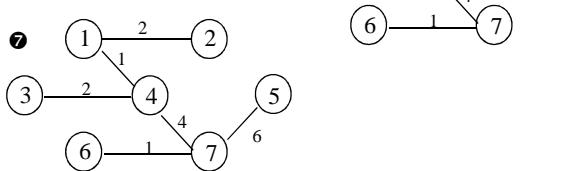
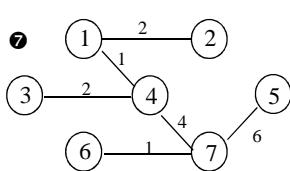
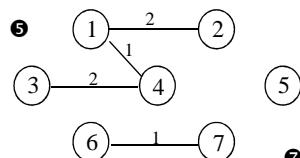
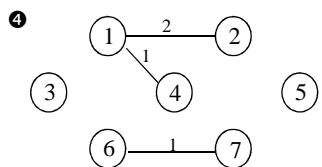
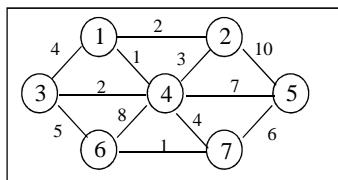
Algoritmo de Kruskal

- analisar as arestas por ordem crescente de peso e aceitar as que não provocarem ciclos (ganancioso)
- método
 - manter uma floresta, inicialmente com um vértice em cada árvore (há $|V|$)
 - adicionar uma aresta é fundir duas ávores
 - quando o algoritmo termina há só uma árvore (de expansão mínima)
- aceitação de arestas – algoritmo de Busca/União em conjuntos disjuntos
 - representados como árvores
 - se dois vértices pertencem à mesma árvore/conjunto, mais uma aresta entre eles provoca um ciclo (2 Buscas)
 - se são de conjuntos disjuntos, aceitar a aresta é aplicar-lhes uma União
- selecção de arestas: ordenar por peso ou, melhor, construir fila de prioridade em tempo linear e usar deleteMin (*heapsort*)
 - tempo no pior caso $O(|E| \log |E|)$, dominado pelas operações na fila
 - como $|E| \leq |V|^2$, $\log |E| \leq 2 \log |V|$, logo eficiência é também $O(|E| \log |V|)$

Evolução do algoritmo de Kruskal



Evolução do algoritmo de Kruskal



Pseudocódigo (Kruskal)

```

void kruskal() {
    int edgesAccepted = 0;

    PriorityQueue<Edge> h = readGraphIntoHeapArray();
    h.buildHeap();
    DisjSet<Vertex> s = new DisjSet(NUM_VERTICES);

    while(edgesAccepted < NUM_VERTICES - 1 ) {
        Edge e = h.deleteMin();      // e = (u,v)
        SetType uset = s.find(u);
        SetType vset = s.find(v);
        if (uset != vset) {
            edgesAccepted++;
            s.union(uset, vset);
        }
    }
}

```

Algoritmos em Grafos: Conectividade

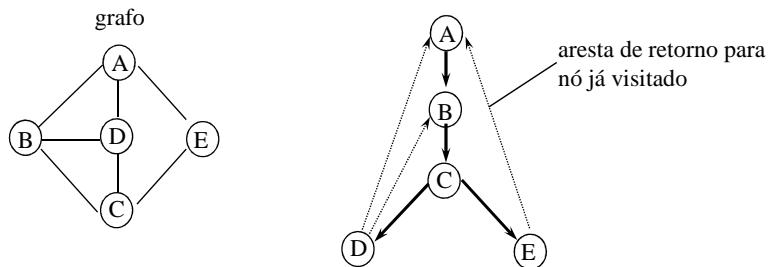
R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

FEUP, MIEIC, CPAL, 2010/2011

Grafos não dirigidos

Coneectividade

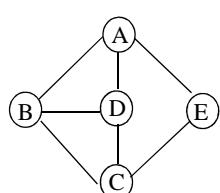
- Um grafo não dirigido é conexo se uma pesquisa em profundidade a começar em qualquer nó visita todos os nós



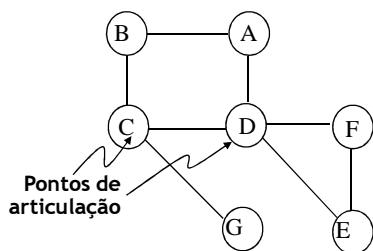
Biconectividade e Pontos de Articulação

- Grafo conexo não dirigido é biconexo se não existe nenhum vértice cuja remoção torne o resto do grafo desconexo
- Pontos de articulação - vértices que tornam o grafo desconexo
- Aplicação - rede com tolerância a falhas

Biconexo



Não biconexo



Algoritmo de detecção de pontos de articulação em tempo linear

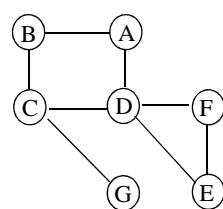
- Início num vértice qualquer
- Pesquisa em profundidade, numerando os vértices ao visitá-los
 - $\text{Num}(v)$, em pré-ordem (antes de visitar adjacentes)
- Para cada vértice v na árvore de visita em profundidade calcular $\text{Low}(v)$, o menor número de vértice que se atinge com zero ou mais arestas na árvore e possivelmente uma aresta de retorno
- Vértice v é ponto de articulação se tiver um filho w tal que $\text{Low}(w) \geq \text{Num}(v)$
- A raiz é ponto de articulação sse tiver mais que um filho na árvore

Cálculo de $\text{Low}(v)$

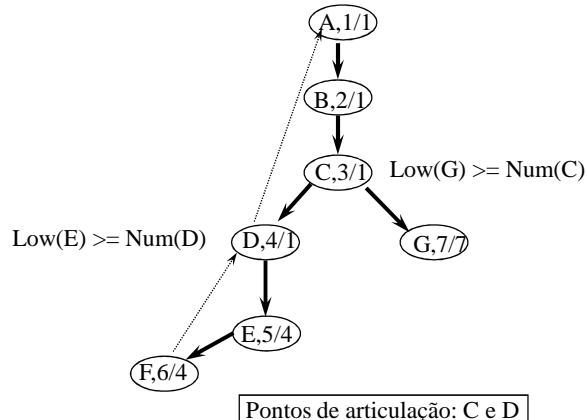
- $\text{Low}(v)$ é mínimo de
 - $\text{Num}(v)$
 - o menor $\text{Num}(w)$ de todas as arestas (v, w) de retorno
 - o menor $\text{Low}(w)$ de todas as arestas (v, w) da árvore
- Na visita em profundidade, inicializa-se $\text{Low}(v)=\text{Num}(v)$ antes de visitar adjacentes, e vai-se actualizando o valor de $\text{Low}(v)$ a seguir a visita a cada adjacente
- Realizável em tempo $O(|E| + |V|)$

Exemplo

Grafo



Uma árvore de expansão em profundidade

 $v, \text{Num}(v)/\text{Low}(v)$ 

Pseudo-código

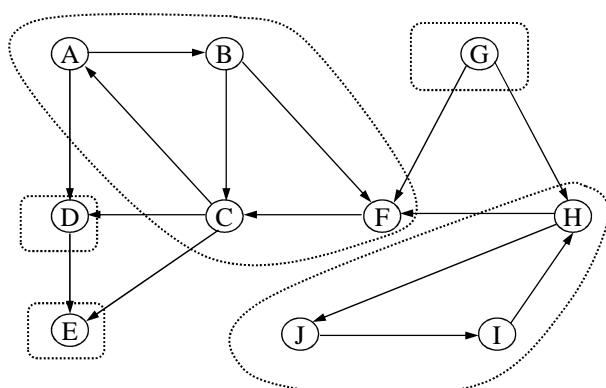
```

// Procura Pontos de Articulação usando dfs
// Contador global e inicializado a 1
void findArt( Vertex v) {
    v.visited = true;
    v.low = v.num = counter++;
    for each w adjacent to v
        if( !w.visited ) { // ramo da árvore
            w.parent = v;
            findArt(w);
            if(w.low >= v.num )
                System.out.println(v,
                    " ponto de articulação ");
                v.low = min(v.low, w.low);
        }
        else
            if ( v.parent != w ) //aresta de retorno
                v.low = min(v.low, w.num );
}

```

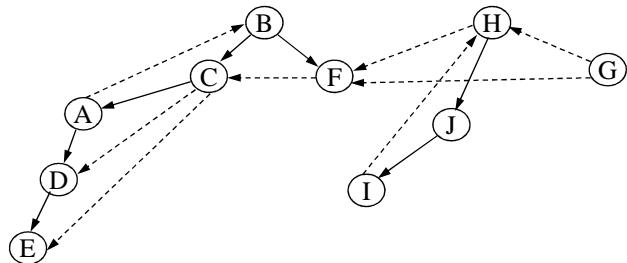
Grafos dirigidos

Componentes fortemente conexos



Árvore de expansão

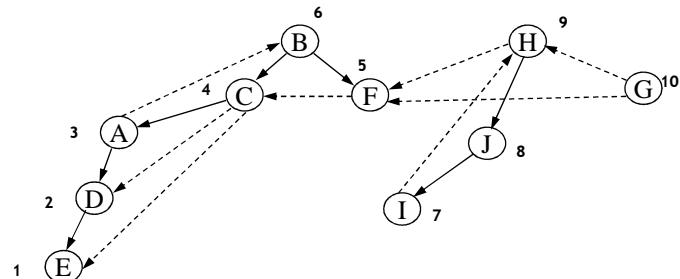
- Pesquisa em profundidade induz uma árvore/floresta de expansão
- Para além das arestas genuínas da árvore, há arestas para nós já marcados
 - arestas de retorno para um antepassado – (A,B), (I,H)
 - arestas de avanço para um descendente – (C,D), (C,E)
 - arestas cruzadas para um nó não relacionado – (F,C), (G,F)



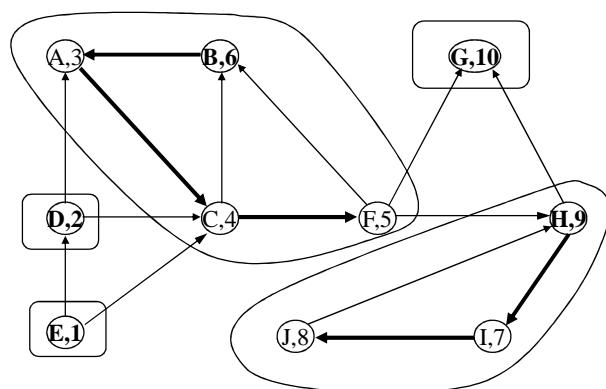
Componentes fortemente conexos

- Método:
 - Pesquisa em profundidade no grafo G determina floresta de expansão, numerando vértices em pós-ordem (ordem inversa de numeração em pré-ordem)
 - Inverter todas as arestas de G (grafo resultante é G_r)
 - Segunda pesquisa em profundidade, em G_r , começando sempre pelo vértice de numeração mais alta ainda não visitado
 - Cada árvore obtida é um componente fortemente conexo, i.e., a partir de um qualquer dos nós pode chegar-se a todos os outros

Numeração em pós-ordem

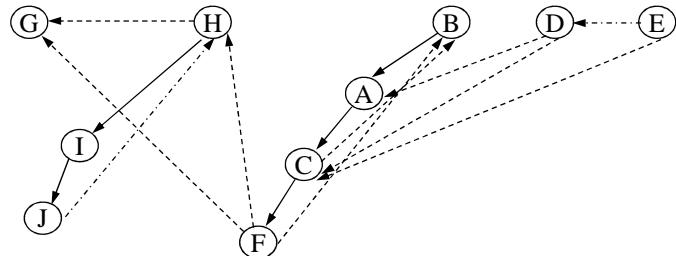


Inversão das arestas e nova visita



G_r : obtido de G por inversão de todas as arestas
Numeração: da travessia de G em pós-ordem

Componentes fortemente conexos



Travessia em pós-ordem de G_r

Componentes fortemente conexos:

{G}, {H, I, J}, {B, A, C, F}, {D}, {E}

* Componentes fortemente conexos

■ Prova

- mesmo componente => mesma árvore de expansão
 - se dois vértices v e w estão no mesmo componente, há caminhos de v para w e de w para v em G e em G_r ; se v e w não pertencerem à mesma árvore de expansão, também não estão no mesmo componente
- mesma árvore de expansão => mesmo componente
 - i.e., há caminhos de v para w e de w para v ou, equivalentemente, se x for a raiz de uma árvore de expansão em profundidade, há caminhos de x para v e de v para x, de x para w e de w para x e portanto entre v e w
 - como v é descendente de x na árvore de G_r , há um caminho de x para v em G_r , logo de v para x em G; como x é a raiz tem o maior número de pós-ordem na primeira pesquisa; portanto, na primeira pesquisa, todo o processamento de v se completou antes de o trabalho em x ter terminado; como há um caminho de v para x, segue-se que v tem que ser um descendente de x na árvore de expansão – caso contrário v terminaria depois de x; isto implica um caminho de x para v em G.

Algoritmos em Grafos: Fluxo Máximo e Fluxo de Custo Mínimo em Redes de Transporte

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

FEUP, MIEIC, CPAL, 2010/2011

Índice

- Conceito de rede de transporte
- Fluxo máximo numa rede de transporte
- Fluxo de custo mínimo numa rede de transporte

Conceito de rede de transporte

Rede de transporte

- Modelar fluxos conservativos entre dois pontos através de canais com capacidade limitada

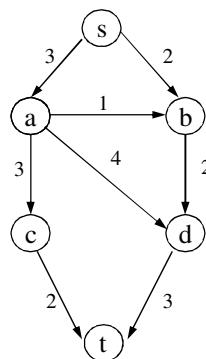
- s: fonte (produtor)
- t: poço (consumidor)
- Fluxo não pode ultrapassar a capacidade da aresta
- Soma dos fluxos de entrada num vértice intermédio igual à soma dos fluxos de saída

- Exemplos:

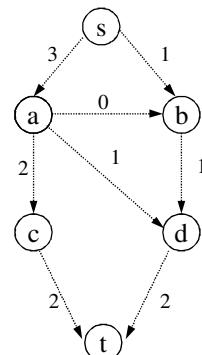
- Rede de abastecimento de líquido ponto a ponto
- Tráfego entre dois pontos

- Em alguns casos, as arestas podem ter custos associados (custo de transportar uma unidade de fluxo)

Rede e capacidades

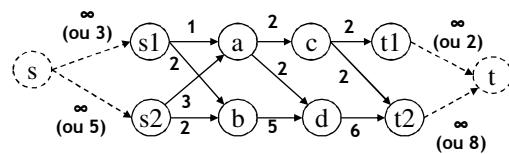


Rede e fluxos



Redes com múltiplas fontes e poços

- Caso de múltiplas fontes e poços (ou mesmo de vértices que podem ser simultaneamente fontes, poços e vértices intermédios) é facilmente redutível ao caso base (uma fonte e um poço)

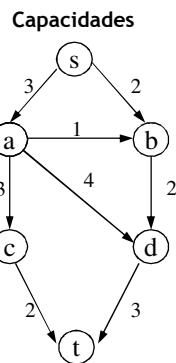


- Se a rede tiver custos nas arestas, as arestas adicionadas têm custo 0

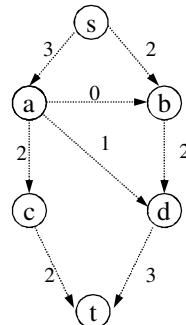
Fluxo máximo numa rede de transporte

Problema

- Encontrar um fluxo de valor máximo (fluxo total que parte de s / chega a t)



Fluxo máximo (valor 5)



Formalização

Dados de entrada :

c_{ij} - capacidade da aresta que vai do nó i a j (0 se não existir)

Dados de saída (variáveis a calcular) :

f_{ij} - fluxo que atravessa a aresta que vai do nó i para o nó j (0 se não existir)

Restrições :

$$0 \leq f_{ij} \leq c_{ij}, \forall_{ij}$$

$$\sum_j f_{ij} = \sum_j f_{ji}, \forall_{i \neq s,t}$$

Objectivo :

$$\max \sum_j f_{sj}$$

Algoritmo de Ford-Fulkerson (1955)

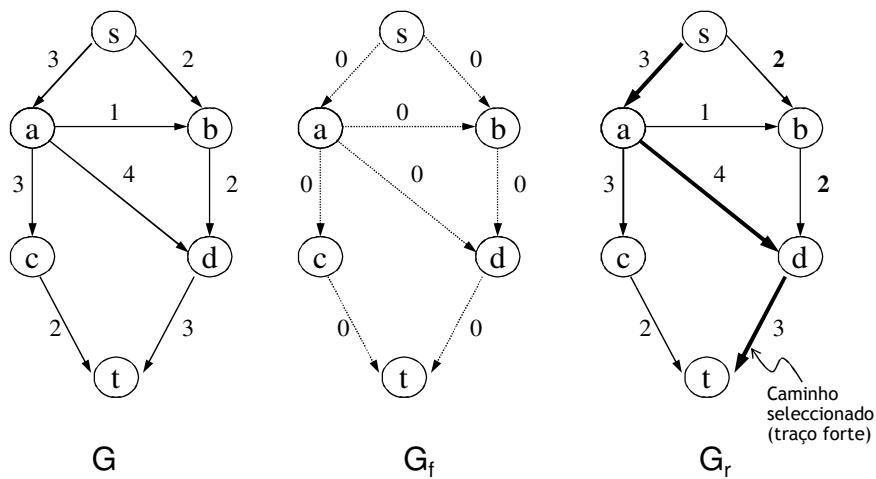
- Estruturas de dados:

- G - grafo base de capacidades $c(v,w)$
- G_f - grafo auxiliar de fluxos $f(v,w)$
 - inicialmente fluxos iguais a 0
 - no fim, tem o fluxo máximo
- G_r - grafo residual (auxiliar)
 - para cada arco (v, w) em G com $c(v,w) > f(v,w)$, cria-se um arco no mesmo sentido em G_r de capacidade igual a $c(v,w) - f(v,w)$ (capacidade disponível)
 - para cada arco (v, w) em G com $f(v,w) > 0$, cria-se um arco em sentido inverso em G_r de capacidade igual a $f(v,w)$
 - arcos necessários para garantir que o algoritmo encontra a solução óptima (ver exemplo)!

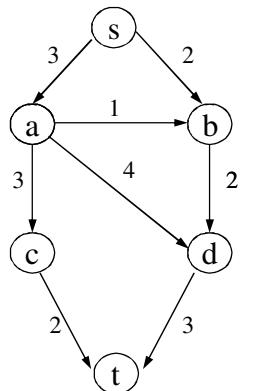
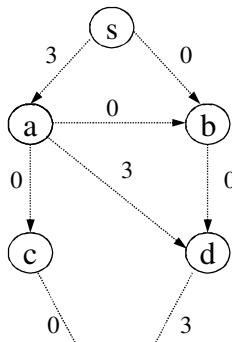
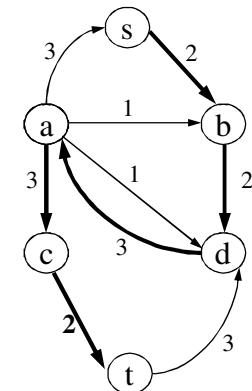
- Método (dos caminhos de aumento):

- Enquanto existirem caminhos entre s e t em G_r
 - Selecionar um caminho qualquer em G_r entre s e t (caminho de aumento)
 - Determinar o valor mínimo (f) nos arcos desse caminho
 - Aumentar esse valor de fluxo (f) a cada um dos arcos respectivos em G_f
 - Recalcular G_r

Exemplo: estado inicial

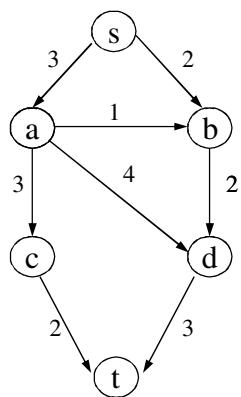
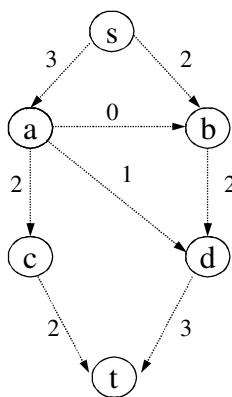
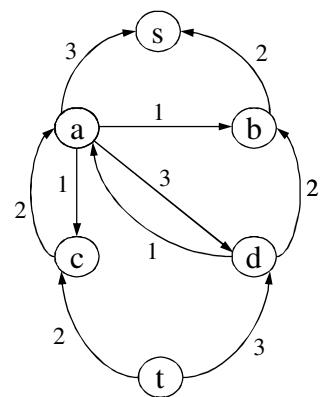


Exemplo: 1^a iteração

 G  G_f  G_r

Se não tivesse as arestas em sentido inverso,
parava aqui com solução não óptima!

Exemplo: 2^a iteração

 G  G_f  G_r

FIM

(valor do fluxo máximo = 5)

Análise do algoritmo de Ford-Fulkerson

- Se as capacidades forem números racionais, o algoritmo termina com o fluxo máximo
- Se as capacidades forem inteiros e o fluxo máximo M
 - Algoritmo tem a propriedade de integralidade: os fluxos finais são também inteiros
 - Bastam M iterações (fluxo aumenta pelo menos 1 por iteração)
 - Cada iteração pode ser feita em tempo $O(|E|)$
 - Tempo de execução total: $O(M |E|)$ - mau

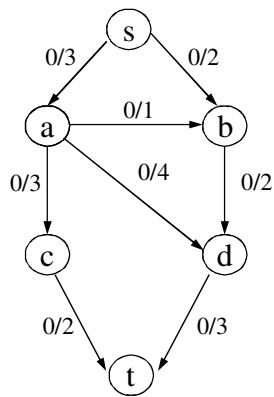
Algoritmo de Edmonds-Karp (1969)

- Em cada iteração do algoritmo de Ford-Fulkerson escolhe-se um caminho de aumento de comprimento mínimo
 - O exemplo apresentado anteriormente já obedece a este critério!
 - Nº máximo de aumentos é $|E| \cdot |V|$ (ver explicação nas referências)
 - Um caminho de aumento mais curto pode ser encontrado em tempo $O(|E|)$ através de pesquisa em largura (ver slides sobre caminhos mais curtos)
 - Tempo de execução: $O(|E|^2 |V|)$
- Interessa apenas como preparação para o algoritmo de Dinic

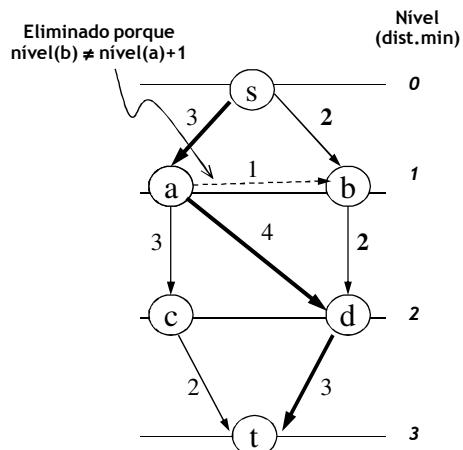
Algoritmo de Dinic (1970)

- Refina algoritmo de Edmonds-Karp, evitando trabalho repetido a achar sucessivos caminhos de aumento de igual comprimento mínimo:
 1. Inicializar os grafos de fluxos (G_f) e de resíduos (G_r) como antes
 2. Calcular o nível de cada vértice, igual à distância mínima a s em G_r
 3. Se $\text{nível}(t) = \infty$, terminar
 4. “Esconder” as arestas (u,v) de G_r em que $\text{nível}(v) \neq \text{nível}(u) + 1$
 - Não podem fazer parte de um caminho mais curto de s para t em G_r !
 - Sem elas, qualquer caminho de s para t em G_r tem comprimento mínimo!
 5. Enquanto existirem caminhos de aumento em G_r (ignorando as arestas escondidas), seleccionar e aplicar um caminho de aumento qualquer
 - Se forem adicionadas a G_r arestas de sentido inverso ao fluxo, ficam também escondidas, pois apenas servem para encontrar caminhos mais compridos
 6. Se $\text{nível}(t) = |V|-1$, terminar; senão saltar para o passo 2 para recalcular os níveis (voltando a considerar todas as arestas de G_r)

Exemplo: estado inicial

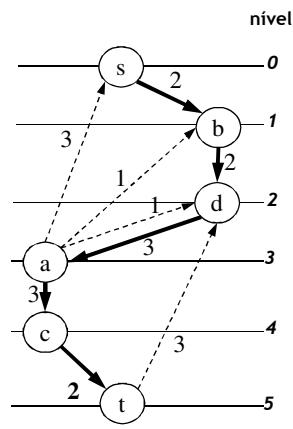
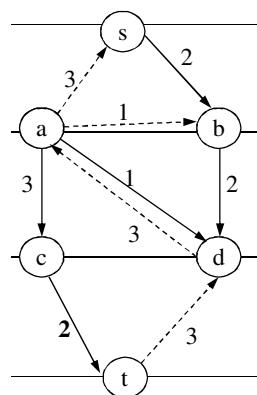
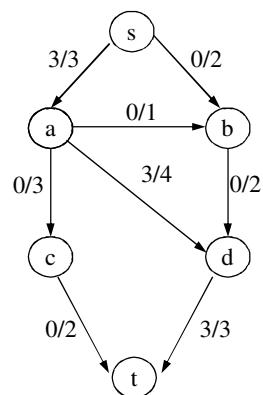


G_f/G



G_r
(nívelado)

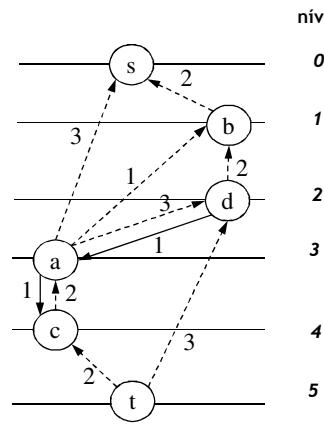
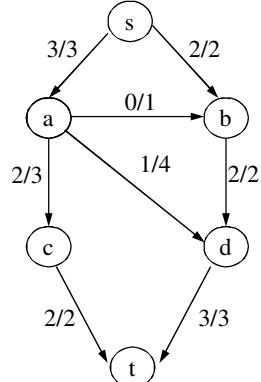
Exemplo: 1^a iteração



Não há mais caminhos de s para t deste nível (comprimento 3) \Rightarrow RENIVELAR



Exemplo: 2^a iteração



Não há mais caminhos de s para t neste nível e nível(t) = $|V| - 1 \Rightarrow$ TERMINAR

Eficiência do algoritmo de Dinic *

- Passo 2

- Cada execução pode ser feita em tempo $O(|E|)$ (assumindo $|E| > |V|$) por uma simples pesquisa em largura (ver slides sobre caminhos mais curtos)
- O nº máximo de execuções é $|V|$, pois cada nova execução só acontece quando se esgotaram os caminhos de aumento de um dado comprimento, e o comprimento dos caminho de aumento só pode crescer até $|V|$

- Passo 5

- O nº máximo de execuções (selecção e aplicação de um caminho de aumento) é o nº máximo de caminhos de aumento, que é o mesmo que no algoritmo de Edmonds-Karp, ou seja, $O(|E|.|V|)$ ($O(|E|)$ para cada comprimento, multiplicado por $|V|$ comprimentos possíveis)
- Cada caminho de aumento pode ser encontrado em tempo $O(|V|)$ no grafo G_r (ignorando as arestas escondidas) por simples pesquisa em profundidade, pois já não há que ter a preocupação de encontrar um caminho mais curto

- Total: $O(|V|^2 |E|)$ (melhoria significativa para grafos densos)

Algoritmo de Dinic em redes unitárias

- Rede unitária:

- Capacidades unitárias
- Todos os vértices excepto s e t têm no máximo uma aresta a entrar ou uma aresta a sair

- Surge em problemas de emparelhamento em grafos bipartidos

- Nesse caso o nº máximo de “renivelamentos” é $|V|^{1/2}$

- Para cada nível/comprimento, os vários caminhos de aumento podem ser seleccionados e aplicados em tempo $O(|E|)$, numa única passagem de visita em profundidade pelo grafo nivelado

- Uma vez que as capacidades são unitárias, as arestas usadas num caminho não têm de voltar a ser consideradas

- Total: $O(|V|^{1/2} |E|)$

* Algoritmos mais eficientes

year	authors	complexity
1955	Ford-Fulkerson [19]	$O(mnU)$
1970	Dinic [15]	$O(mn^2)$
1969	Edmonds-Karp [17]	$O(m^2n)$
1972	Dinic [15], Edmonds-Karp [17]	$O(m^2 \log U)$
1973	Dinic [16], Gabow [20]	$O(mn \log U)$
1974	Karzanov [37]	$O(n^3)$
1977	Cherkassky [11]	$O(n^2 m^{1/2})$
1980	Galil-Naamad [21]	$O(mn(\log n)^2)$
1983	Sleator-Tarjan [44]	$O(mn \log n)$
1986	Goldberg-Tarjan [26]	$O(mn \log(n^2/m))$
1987	Ahuja-Orlin [3]	$O(mn + n^2 \log U)$
1987	Ahuja Orlin Tarjan [4]	$O(mn \log(2 + n \sqrt{\log U/m}))$
1990	Cheriyan-Hagerup-Mehlhorn [9]	$O(n^3 / \log n)$
1990	Alon [5]	$O(mn + n^{4/3} \log n)$
1992	King-Rao-Tarjan [38]	$O(mn + n^{2+\epsilon})$
1993	Phillips-Westbrook [42]	$O(mn \log_{m/n} n + n^2 (\log n)^{2+\epsilon})$
1994	King-Rao-Tarjan [39]	$O(mn \log_{m/(n \log n)} n)$
1997	Goldberg-Rao [23]	$O(\min\{m^{1/2}, n^{2/3}\} m \log(n^2/m) \log U)$

($m = |E|$, $n = |V|$, $U =$ capacidade máxima)

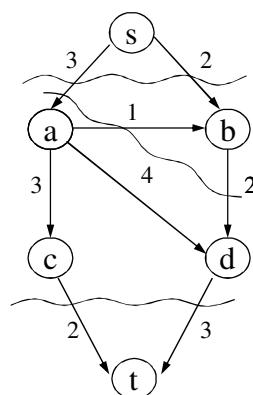
T. Asano and Y. Asano: recent developments in Maximum Flow Algorithms. Journal of the Operations Research, 1 (2000).



* Dualidade entre fluxo máximo e corte mínimo

- Teorema: O valor do fluxo máximo numa rede de transporte é igual à capacidade do corte mínimo
 - Um corte (S, T) numa rede de transporte $G=(V,E)$ com fonte s e poço t é uma partição de V em conjuntos S e $T=V-S$ tal que $s \in S$ e $t \in T$
 - A capacidade de um corte (S, T) é a soma das capacidades das arestas cortadas dirigidas de S para T
 - Um corte mínimo é um corte cuja capacidade é mínima

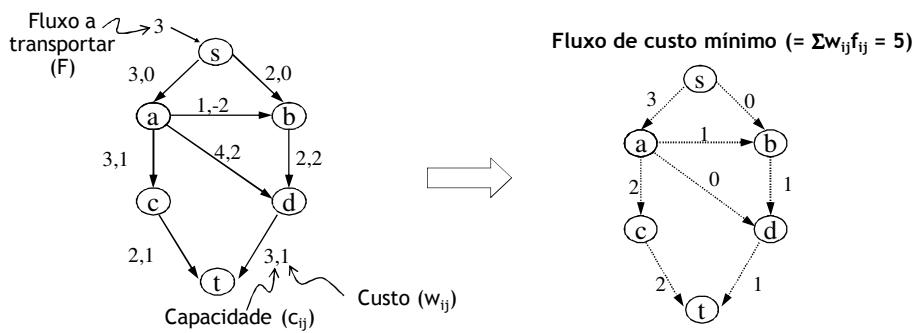
Cortes mínimos na rede do exemplo:



* Fluxo de custo mínimo

Problema

- O objectivo é transportar uma certa quantidade F de fluxo (\leq máximo permitido pela rede) da fonte para o poço, com um custo total mínimo
 - Para além da capacidade, arestas têm associado um custo (custo de transportar uma unidade de fluxo)
 - Podem existir arestas de custo negativo (útil em problemas de maximização do valor, introduzindo sinal negativo)



Formalização

Dados de entrada :

c_{ij} - capacidade da aresta que vai do nó i a j (0 se não existir)

w_{ij} - custo de passar uma unidade de fluxo pela aresta (i, j)

F - quantidade de fluxo a passar pela rede

Dados de saída (variáveis a calcular):

f_{ij} - fluxo que atravessa a aresta que vai do nó i para o nó j (0 se não existir)

Restrições:

$$0 \leq f_{ij} \leq c_{ij}, \forall ij$$

$$\sum_j f_{ij} = \sum_j f_{ji}, \forall i \neq s, t$$

$$\sum_j f_{sj} = F$$

Objectivo :

$$\min \sum_{ij} f_{ij} \times w_{ij}$$

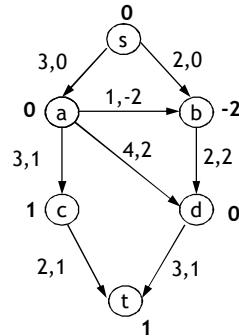
Método dos caminhos de aumento mais curtos

- Algoritmo ganancioso: no algoritmo de Ford-Fulkerson, escolhe-se em cada momento um caminho de aumento de custo mínimo
 - Pára-se quando se atinge o fluxo pretendido ou quando não há mais caminhos de aumento (neste caso dá um **fluxo máximo de custo mínimo**)
- Restrição: aplicável só quando a rede não tem ciclos de custo negativo
 - Senão aplica-se método mais genérico (cancelamento de ciclos negativos)
- Prova-se que dá a solução óptima (ver referências)
- Dificuldade: arestas de custo negativo no grafo de resíduos
 - Devido a custos iniciais negativos ou a inversão de arestas no grafo de resíduos
 - Obriga a usar algoritmo menos eficiente na procura do caminho de custo mínimo
- Solução: conversão do grafo de resíduos num equivalente (para efeito de encontrar caminho de custo mínimo) sem custos negativos

Conversão do grafo de resíduos (1/2)

1. Determinar a distância mínima de s a todos os vértices no grafo de resíduos ($d(v)$)

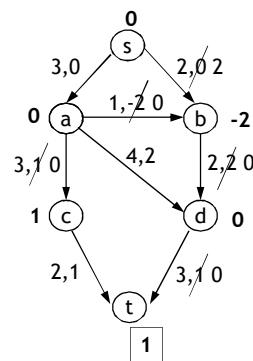
- Grafo de resíduos inicial (com fluxos nulos) é idêntico ao grafo de capacidades
- Se existirem arestas (mas não ciclos) de peso negativo no grafo de resíduos inicial, usa-se o algoritmo de Bellman-Ford, de tempo $O(|E| |V|)$



Conversão do grafo de resíduos (2/2)

2. Substituir os custos iniciais $w(u,v)$ por custos “reduzidos” $w'(u,v) = w(u,v) + d(u) - d(v)$

- $w'(u,v) \geq 0$ pois $d(v) \leq d(u) + w(u,v)$
- O custo w' de um caminho de s a t , usando os custos reduzidos, é igual ao custo usando os custos antes da redução subtraído de $d(t)$ (demonstrar !)

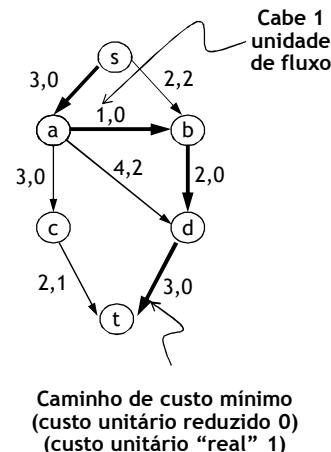


Próximo custo é 1

Determinação do próximo caminho de aumento

3. Selecionar um caminho de custo mínimo de s para t no grafo de resíduos

- Os caminhos de custo mínimo de s para t têm custo reduzido 0 e custo “real” (antes da redução) $d(t)$
- Como os caminhos de custo mínimo percorrem apenas arestas de custo 0, podem ser encontrados como uma pesquisa simples em tempo linear



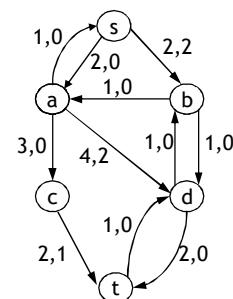
Aplicação do caminho de aumento

4. Aplicar o caminho de aumento

- Custo das arestas invertidas no grafo de resíduos é multiplicado por -1 (*)

$$\begin{aligned} (*) \text{ custo aresta invertida: } \\ w'(v,u) &= w(v,u)+d(v)-d(u) \\ &= -w(u,v)+d(v)-d(u) \\ &= -w'(u,v) \end{aligned}$$

- Só que $-1 \times 0 = 0 \dots$
- Evita-se assim a introdução de arestas de custo negativo!

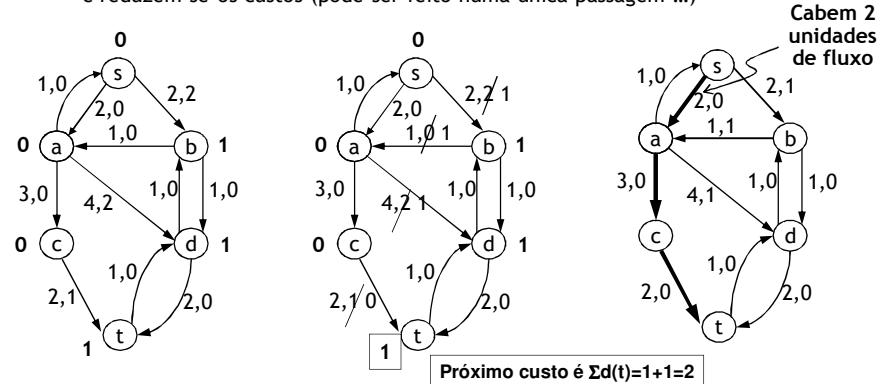


(quantidade actual do fluxo = 1)
(custo actual do fluxo = 1)

Nova conversão do grafo de resíduos

5. Como não há mais caminhos de aumento de custo 0, volta-se a efectuar uma “redução” dos custos no grafo de resíduos

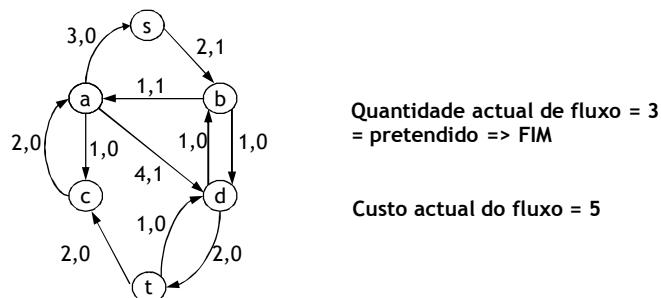
- Isto é, recalcularam-se as distâncias mínimas (agora pelo algoritmo de Dijkstra!), e reduzem-se os custos (pode ser feito numa única passagem ...)



Repetição do processo

6. Aplicar o caminho de aumento de custo 0

- Actualiza-se o grafo de resíduos



Eficiência *

- Primeira redução do grafo de resíduos: $O(|V| |E|)$ pelo algoritmo de Bellman-Ford
- Subsequentes reduções do grafo de resíduos e determinação do caminho de aumento de custo mínimo: $O(|E| \log |V|)$ pelo algoritmo de Dijkstra
- Se todas as grandezas forem inteiras, o nº máximo de iterações é F ($\leq |E|$), pois em cada iteração o valor do fluxo é incrementado de uma unidade
- Tempo total fica $O(F |E| \log |V|)$ usando heap binário (2-heap)
- Pode ser melhorado para $O(F |E| \log_{\lceil |E| / |V| + 1 \rceil} |V|)$ usando d-heap em que $d = \lceil |E| / |V| + 1 \rceil$ (nº de filhos de cada nó do heap), de acordo com o seguinte teorema:
 - Teorema (Johnson 1977): Seja $d = \lceil m/n+1 \rceil$. Um d-heap suporta m operações do tipo 1 (insert/decreaseKey) e n operações de tipo 2 (delete) em tempo $O(m \log_d n)$ (neste caso fazemos $n=|V|$ e $m=|E|$)

Referências e informação adicional

- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
- “Efficient algorithms for shortest paths in sparse networks”. D. Johnson, J. ACM 24, 1 (Jan. 1977), 1-13

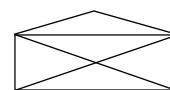
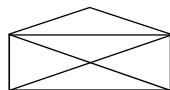
Algoritmos em Grafos: Circuitos de Euler e Problema do Carteiro Chinês

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

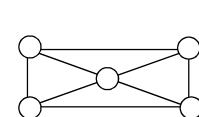
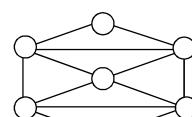
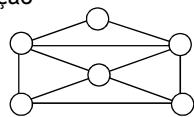
FEUP, MIEIC, CPAL, 2010/2011

Circuitos de Euler

- Puzzle: desenhar as figuras abaixo sem levantar o lápis e sem repetir arestas; de preferência, terminando no mesmo vértice em que iniciar.



- Reformulação como problema em Teoria de Grafos: pôr um vértice em cada intersecção



- Caminho de Euler:** caminho que visita cada aresta exactamente uma vez
- Problema resolvido por Euler em 1736 e que marca o início da Teoria dos Grafos
- Círculo de Euler:** caminho de Euler que começa e acaba no mesmo vértice

Condições necessárias e suficientes

- Um grafo *não dirigido* contém um *círculo de Euler* sse
 - (1) é conexo e
 - (2) cada vértice tem grau (nº de arestas incidentes) par.
- Um grafo *não dirigido* contém um *caminho de Euler* sse
 - (1) é conexo e
 - (2) todos menos dois vértices têm grau par (estes dois vértices serão os vértices de início e fim do caminho).
- Um grafo *dirigido* contém um *círculo de Euler* sse
 - (1) é (fortemente) conexo e
 - (2) cada vértice tem o mesmo grau de entrada e de saída.
- Um grafo *dirigido* contém um *caminho de Euler* sse
 - (1) é (fortemente) conexo e
 - (2) todos menos dois vértices têm o mesmo grau de entrada e de saída, e os dois vértices têm graus de entrada e de saída que diferem de 1.

Método baseado em pesquisa em profundidade para encontrar um círculo de Euler

- **Método:**
 1. Escolher um vértice qualquer e efectuar uma pesquisa em profundidade a partir desse vértice (se o grafo satisfizer as condições necessárias e suficientes, esta pesquisa termina necessariamente no vértice de partida, formando um círculo, embora não necessariamente de Euler)
 2. Enquanto existirem arestas por visitar
 - 2.1 Procurar o primeiro vértice no caminho (círculo) obtido até ao momento que possua uma aresta não percorrida
 - 2.2 Lançar uma sub-pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas)
 - 2.3 Inserir o resultado (círculo) no caminho principal
- **Tempo de execução:** $O(|E| + |V|)$
 - Cada vértice e aresta é percorrido uma única vez
 - Cada vez que se percorre um adjacente, avança-se o apontador de adjacentes (para não voltar a percorrer as mesmas arestas)
 - Usam-se listas ligadas para efectuar inserções em tempo constante

Exemplo em grafo não dirigido

Arestas por visitar	Caminho desta iteração	Caminho acumulado
 1 ^a iter.	1-3*-2-1-6-7-1 Com arestas por visitar	1-3*-2-1-6-7-1
 2 ^a iter.	3-4-5-3	1-3-4-5-3-2-1-6-7-1 (Círculo de Euler)



Problema do carteiro chinês (*Chinese postman problem*)

- Dado um grafo pesado conexo $G=(V,E)$, encontrar um caminho fechado (i.e., com início e fim no mesmo vértice) de peso mínimo que atravesse cada aresta de G pelo menos uma vez.
 - A um caminho nessas condições chama-se *percurso óptimo do carteiro Chinês*.
 - A qualquer caminho fechado (não necessariamente de peso mínimo) que atravesses cada aresta de G pelo menos uma vez chama-se *percurso do carteiro*.
- Problema estudado pela primeira vez p/ Mei-Ku Kuan em 1962, relacionado com a distribuição de correspondência ao longo de um conjunto de ruas, partindo e terminando numa estação de correios.
- Se o grafo G for Euleriano, então qualquer circuito de Euler é um percurso óptimo do carteiro Chinês.
- Se o grafo G não for Euleriano, pode-se construir um grafo Euleriano G^* duplicando algumas arestas de G , seleccionadas de forma a conseguir um grafo Euleriano com peso total mínimo - ver a seguir. Nunca é necessário visitar cada aresta mais do que duas vezes!



Algoritmo para achar um percurso óptimo do carteiro chinês num grafo não dirigido

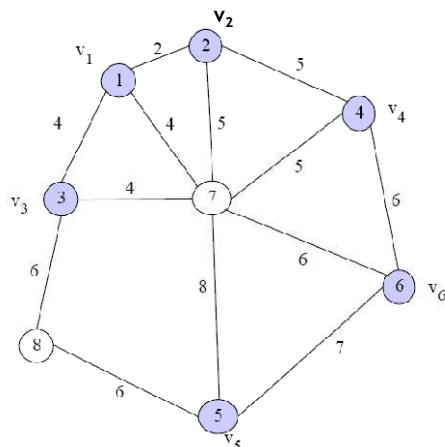
1. Achar todos os vértices de grau ímpar (com nº ímpar de arestas incidentes) em G . Seja k o nº (par!) destes vértices. Se $k=0$, fazer $G^*=G$ e saltar para o passo 6.
2. Achar os caminhos mais curtos (e distâncias mínimas) entre todos os pares de vértices de grau ímpar em G .
3. Construir um grafo completo G' com os vértices de grau ímpar de G ligados entre si por arestas de peso igual à distância mínima calculada no passo 2.
4. Encontrar um emparelhamento perfeito de peso mínimo em G' (ver a seguir). Isto corresponde a emparelhar os vértices de grau ímpar de G , por forma a minimizar a soma das distâncias entre vértices emparelhados.
5. Para cada par (u, v) no emparelhamento perfeito encontrado, adicionar pseudo-arestas (arestas paralelas duplicadas) a G ao longo de um caminho mais curto entre u e v . Seja G^* o grafo resultante.
6. Achar um circuito de Euler em G^* . Este circuito é um percurso óptimo do carteiro Chinês.

Realização do passo 4

- Passo mais complexo
- Um emparelhamento perfeito é um emparelhamento que envolve todos os vértices
- O problema de encontrar um emparelhamento perfeito de peso mínimo pode ser reduzido ao problema de encontrar um emparelhamento de peso máximo num grafo genérico por uma simples mudança de pesos
 - Basta substituir cada peso w_{ij} por $M+1-w_{ij}$, em que M é o peso da aresta mais pesada
 - Sendo o grafo completo e com número par de vértices, um emparelhamento de peso máximo é necessariamente perfeito
- Um emparelhamento de peso máximo num grafo genérico pode ser encontrado em tempo polinomial - ver referências

Exemplo (1/4)

- Grafo G e vértices de grau ímpar (sombreados)



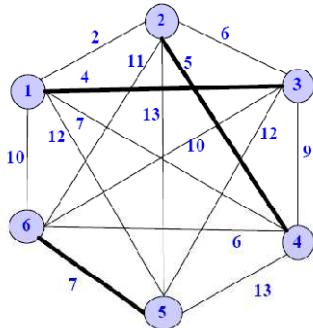
Exemplo (2/4)

- Distâncias (pelo caminho mais curto) entre todos os pares de vértices de grau ímpar

$d(v_i, v_j)$	$v1$	$v2$	$v3$	$v4$	$v5$	$v6$
$v1$	-	2	4	7	12	10
$v2$		-	6	5	13	11
$v3$			-	9	12	10
$v4$				-	13	6
$v5$					-	7
$v6$						-

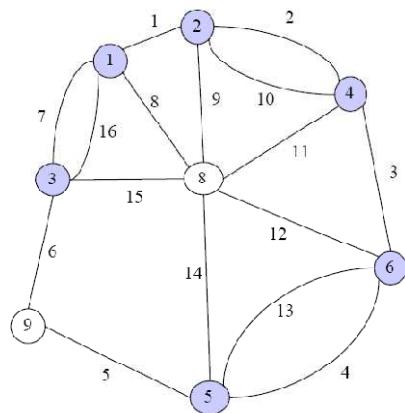
Exemplo (3/4)

- Grafo G' correspondente (com vértices unidos por arestas de peso igual à distância) e emparelhamento perfeito de peso mínimo (arestas a traço forte):



Exemplo (4/4)

- Grafo G^* correspondente, com uma possível numeração das arestas ao longo de um circuito de Euler (distâncias não são mostradas):

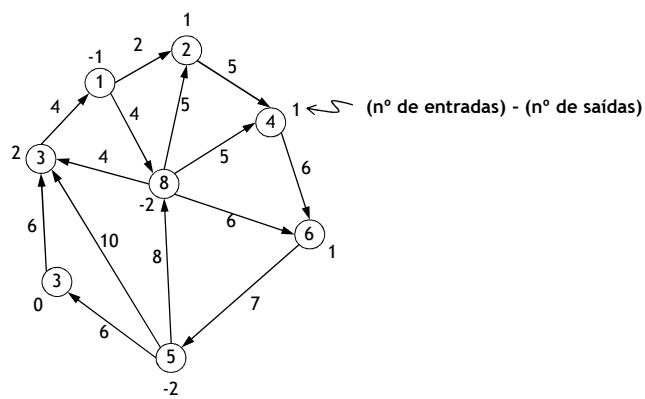


* Caso de grafos dirigidos

- Modificações ao algoritmo anterior:
 - 1) Identificam-se os vértices com nº diferentes de arestas a entrar e a sair
 - 2) Procuram-se os caminhos mais curtos de vértices que têm défice de saídas para vértices que têm défices de entradas
 - 3) Constrói-se um grafo bipartido, em que os vértices são distinguidos consoante têm mais arestas a entrar ou a sair e são replicados consoante a diferença entre entradas e saídas
 - 4) Procura-se um emparelhamento perfeito de peso mínimo num grafo bipartido
 - Em vez de replicar os vértices no ponto 3, pode-se associar a cada vértice uma multiplicidade ($nº$ de emparelhamentos em que pode participar) e converter o problema directamente para um problema de fluxo máximo de custo mínimo em que algumas arestas têm capacidade superior a 1 -> Ver no exemplo a seguir
- Resolúvel igualmente em tempo polinomial
- Infelizmente, o problema é NP-completo (tempo exponencial) quando se combinam arestas dirigidas com arestas não dirigidas (grafos mistos)
 - Exemplo: determinar o percurso a seguir pelo camião do lixo, quando algumas ruas têm sentidos únicos

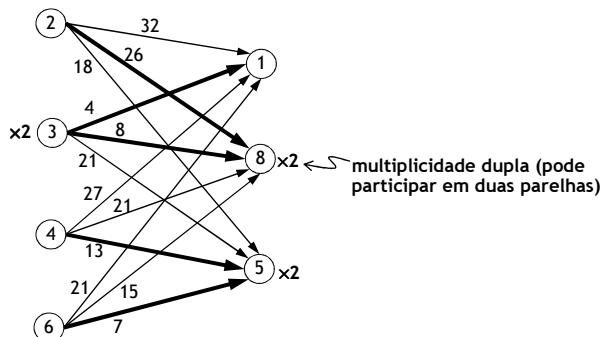
* Exemplo (1/4)

- Grafo G e vértices com diferente $nº$ de entradas e saídas:



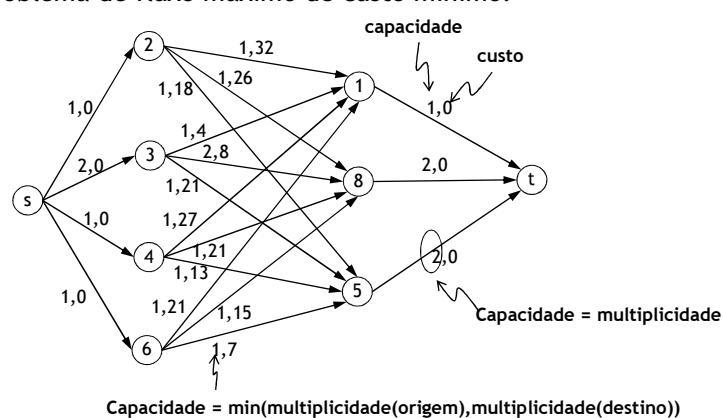
* Exemplo (2/4)

- Grafo G' com distâncias de vértices com deficit de saídas para vértices com deficit de entradas, e emparelhamento perfeito de peso mínimo (arestas a traço forte, obtidas resolvendo o problema de fluxo máximo de custo mínimo indicado no slide seguinte):



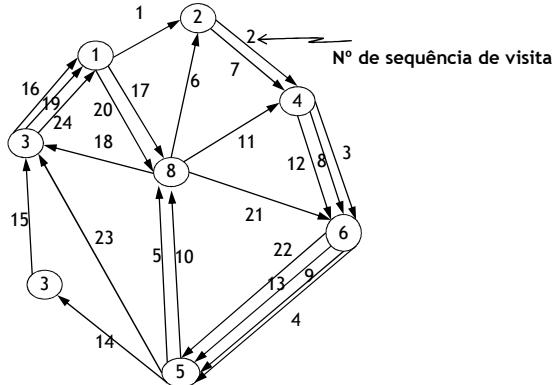
* Exemplo (3/4)

- Formulação do problema de emparelhamento óptimo como problema de fluxo máximo de custo mínimo:



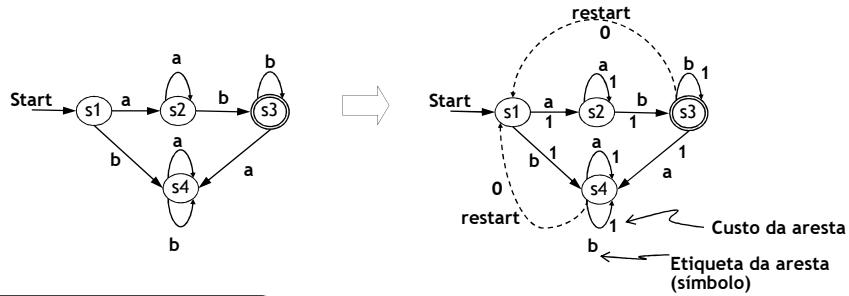
* Exemplo (4/4)

- Grafo G^* correspondente, após duplicação de caminhos mais curtos entre os vértices emparelhados, e uma numeração possível das arestas ao longo de um circuito de Euler (distâncias não são mostradas):



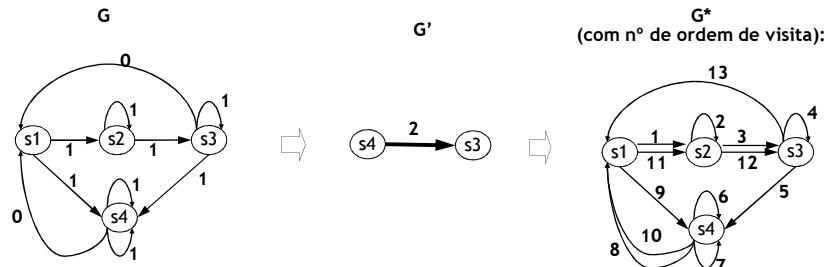
* Exemplo de aplicação (1/2)

- Achar um conjunto de sequências de teste completas (do estado inicial a um estado final) de comprimento total mínimo cobrindo todas as transições num autómato finito
 - Ligam-se os estados finais ao estado inicial e procura-se um percurso óptimo do carteiro
 - Nota: conceito de estado final faz mais sentido em máquinas de estados UML; no caso de autómatos finitos, podem-se considerar como tal estados de aceitação e estados absorventes (onde não é possível sair)



* Exemplo de aplicação (2/2)

- Resolução do problema do carteiro chinês dirigido:



- Solução final:

- Caminho de Euler usando etiquetas:
a-a-b-b-a-a-b-restart-b-restart-a-b-restart
- Strings de teste: aabbaab, b, ab

Referências e mais informação

- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998

Algoritmos em Grafos: Emparelhamentos (*matching*) e Casamentos Estáveis (*stable marriage*)

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

FEUP, MIEIC, CPAL, 2010/2011

Nota prévia

- Em geral, apenas são abordados os slides que descrevem os problemas de emparelhamento, mas não os slides que descrevem os algoritmos que resolvem esses problemas
- Os slides marcados com asterisco não são abordados nas aulas, mas servem como consulta e referência para aprofundar a matéria pelo estudante

Índice

- Emparelhamentos
 - Emparelhamentos de tamanho máximo em grafos bipartidos
 - Emparelhamentos de peso máximo em grafos bipartidos
 - Emparelhamentos de tamanho máximo em grafos genéricos
 - Emparelhamentos de peso máximo em grafos genéricos
- Casamentos estáveis
 - Com ordem estrita de preferências e listas de preferências completas
 - Com ordem estrita de preferências e listas de preferências incompletas
 - Aplicação à colocação de professores

Emparelhamentos

Conceito de emparelhamento

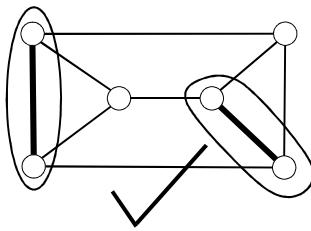
- Seja o grafo não dirigido $G = (V, E)$
- Formalmente, um emparelhamento (*matching*) M em G é um conjunto de arestas que não contém mais do que uma aresta incidente no mesmo vértice
- Ou: um conjunto de arestas independentes num grafo G é um conjunto de arestas sem vértices comuns!

Alguns conceitos

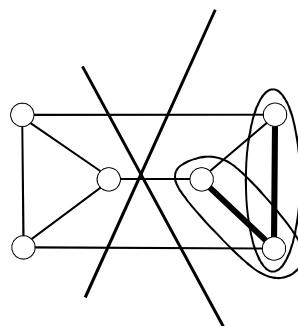
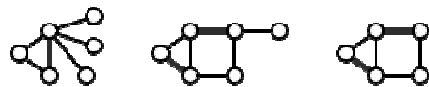
- Um vértice u é dito estar **emparelhado** (ou **saturado**) se tem uma aresta incidente no emparelhamento
- Um **emparelhamento maximal** é um emparelhamento M em G com a seguinte propriedade:
 - Se qualquer aresta não pertencente a M é adicionada a M , M não é mais um emparelhamento de G ; ou seja, M não é propriamente um subconjunto de qualquer outro emparelhamento no grafo G . Em outras palavras, um emparelhamento M é maximal se toda aresta em G tem uma intersecção não vazia com uma aresta de M
- Um **emparelhamento máximo** é um emparelhamento M em G que contém o número máximo de arestas. Poderá haver muitos emparelhamentos que conformam esta propriedade. O número $v(G)$ é o tamanho do emparelhamento máximo.
- Todo emparelhamento máximo é maximal
- Um **emparelhamento** é dito **perfeito** se inclui todos os vértices do grafo.

Conceito de emparelhamento

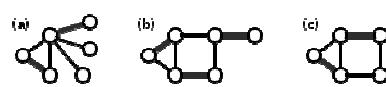
- Exemplo:



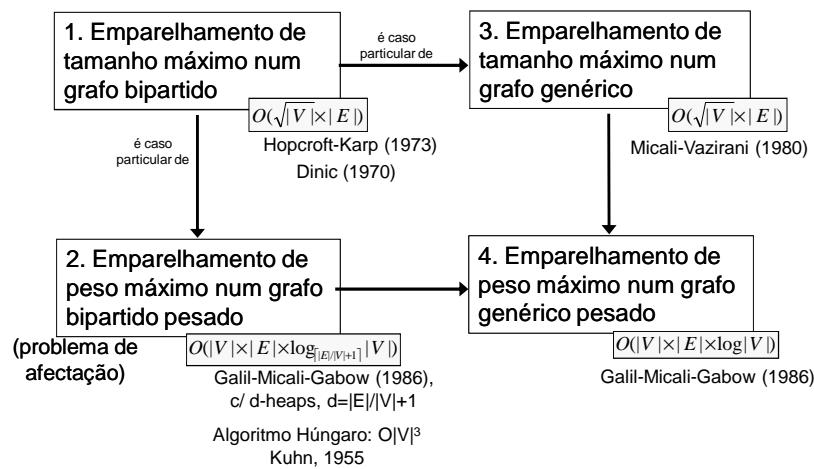
• Emparelhamento Maximal



• Emparelhamento Máximo

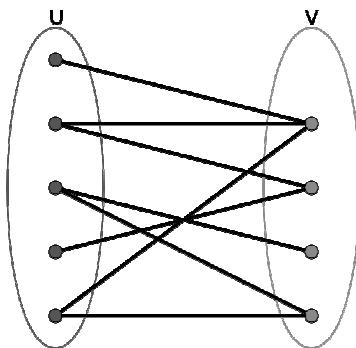


Problemas de emparelhamento



Grafos bipartidos

- Um grafo G é dito ser **bipartido** (ou bigrafo) se os seus vértices podem ser divididos em dois conjuntos disjuntos U e V , tal que toda aresta em G liga um vértice u , em U , a um vértice v , em V .

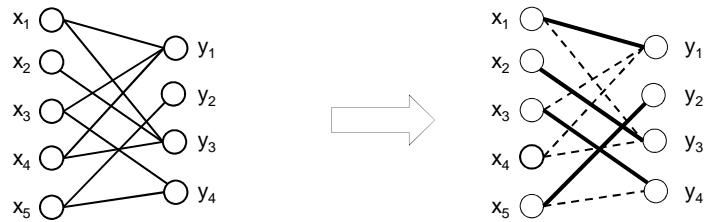


Redução a problemas em redes de transporte

- Problemas de **emparelhamento em grafos bipartidos** são redutíveis a problemas em redes de transporte (com capacidades unitárias)
 - Emparelhamento de tamanho máximo \rightarrow fluxo máximo
 - Emparelhamento de peso máximo \rightarrow fluxo de custo mínimo (custo = -peso)
- Grafos genéricos sem ciclos de tamanho ímpar são redutíveis a grafos bipartidos
 - Basta fazer uma pesquisa em largura, a qual gera uma floresta de pesquisa em largura (ver slides sobre pesquisa em largura), e separar depois os vértices de profundidade par dos vértices de profundidade ímpar nessa floresta
- Grafos genéricos com ciclos de tamanho ímpar exigem algoritmos mais elaborados

1. Emparelhamento de tamanho máximo num grafo bipartido

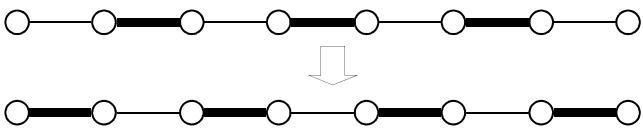
pessoas candidatam-se a empregos



Grafo bipartido - o conjunto de vértices pode ser partido em dois conjuntos X e Y tal que todas as arestas têm um extremo em X e outro em Y

Quais são as outras soluções?

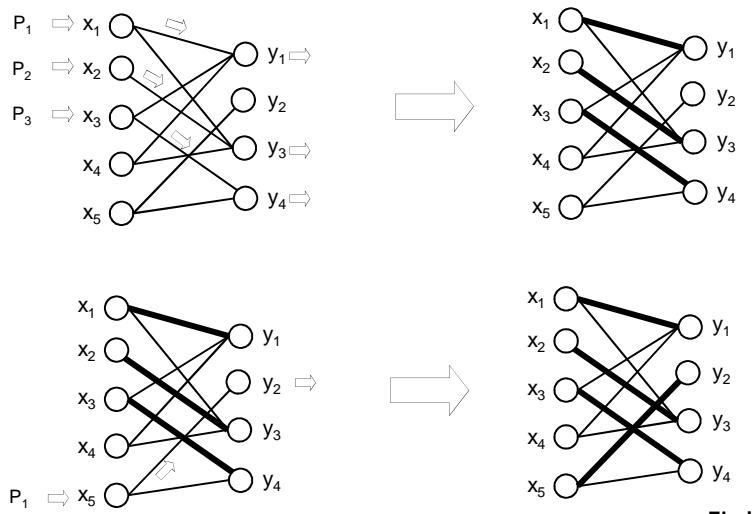
* Método dos caminhos de aumento (augmenting paths)

- Definição: Um caminho simples P (aberto ou fechado) num grafo G (bipartido ou genérico) é um *caminho de aumento* em relação a um emparelhamento M se
 - começa e acaba em vértices que não estão em M (vértices livres), e
 - passa alternadamente em arestas que não pertencem a M e pertencem a M
- Removendo de M as arestas de P que pertencem a M , e adicionando as que não pertencem a M , aumenta-se o tamanho do emparelhamento de 1
 
- A aplicação sucessiva desta técnica, partindo de um emparelhamento vazio, até não existirem mais caminhos de aumento, dá um emparelhamento de tamanho máximo (num grafo bipartido ou genérico), de acordo com o ...
- Teorema de Berge: Um emparelhamento M num grafo G (bipartido ou genérico) tem tamanho máximo sse não existir nenhum caminho de aumento

* Algoritmo de Hopcroft-Karp (1973)

- **Algoritmo:**
 1. $M \leftarrow \{\}$
 2. Encontrar um conjunto maximal de caminhos de aumento de comprimento mínimo, sem vértices em comum
 - Um conjunto maximal é um conjunto a que não é possível acrescentar mais elementos
 - Minimizando o comprimento dos caminhos, maximiza o nº de caminhos e portanto o aumento de M
 - Algoritmo proposto por Hopcroft-Karp para este passo é basicamente o algoritmo de Dinic
 3. Se não existir nenhum caminho de aumento, terminar
 4. Remover de M as arestas dos caminhos de aumento que pertenciam a M , e adicionar a M as arestas dos caminhos de aumento que não pertenciam em M
 5. Continuar no passo 2
- Constatou-se posteriormente que o algoritmo de Hopcroft-Karp (que inclui mais detalhes sobre a realização do passo 2) é basicamente equivalente ao algoritmo de Dinic (fluxo máximo)!
- **Eficiência:**
 - Cada execução do passo 2 pode ser efectuada em tempo $O(|E|)$
 - Hopcroft-Karp provaram que o nº máximo de iterações (fases) é $V^{1/2}$
 - Multiplicando: $O(\sqrt{V} \times |E|)$

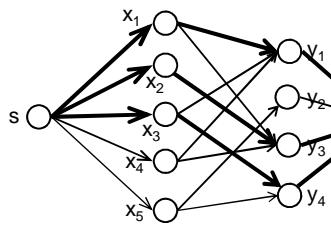
* Exemplo de aplicação do algoritmo de Hopcroft-Karp



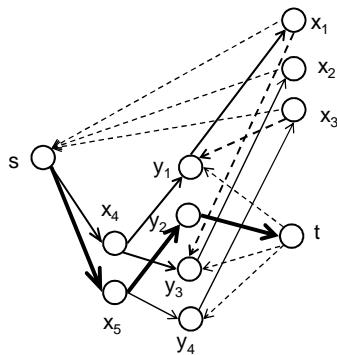
Fim!

Resolução como problema de fluxo máximo numa rede unitária (Dinic)

Rede correspondente com todas as capacidades unitárias.
É também o grafo inicial de resíduos.
Está já ordenado por níveis da esquerda para a direita
Caminhos de aumento marcados a traço forte.



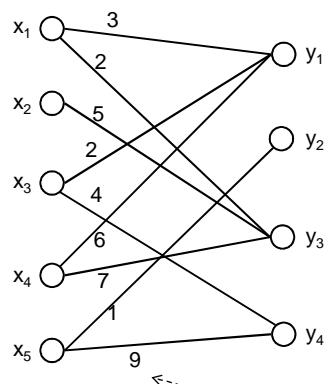
Novo grafo de resíduos, ordenado por níveis da esquerda para a direita.
Caminho de aumento marcado a traço forte.



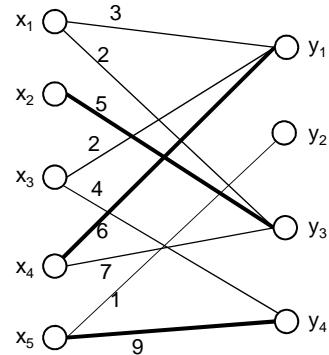
Depois de aplicar este caminho de aumento, no novo grafo de resíduos t não é alcançável, pelo que termina!

2. Emparelhamento de peso máximo num grafo bipartido pesado (problema de afectação)

pessoas candidatam-se a empregos



(não ter aresta é o mesmo que ter peso negativo)



Peso total: 20

(estratégia gananciosa daria 19)

* Representação tabular (matriz de adjacências)

	y1	y2	y3	y4
x1	3	-	2	-
x2	-	-	5	-
x3	2	-	-	4
x4	6	-	7	-
x5	-	1	-	9

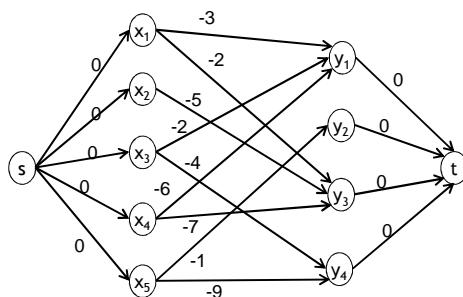
Pode-se colocar um valor negativo (-1) para garantir que não é seleccionado

Mais adequada quando o grafo é denso!

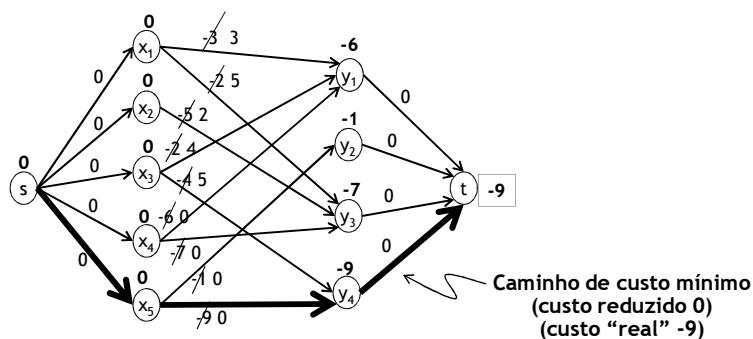
Se o grafo for denso, um bom algoritmo é o algoritmo Húngaro!

* Resolução como problema de fluxo de custo mínimo

- Capacidades unitárias (como tal apenas se mostram os custos e não as capacidades)
- Faz-se custo no problema de transporte = simétrico do peso no problema de emparelhamento
 - Origina arestas de custo negativo, mas não há ciclos
- Aplica-se método dos caminhos de aumento de custo mínimo, parando-se quando o próximo caminho de aumento tem custo real ≥ 0

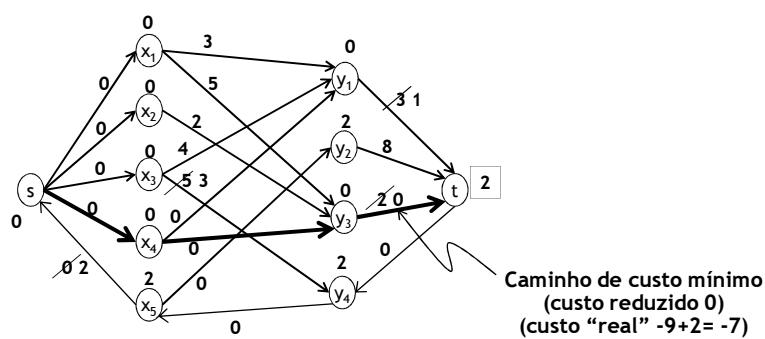


* Resolução como problema de fluxo de custo mínimo - 1^a iteração



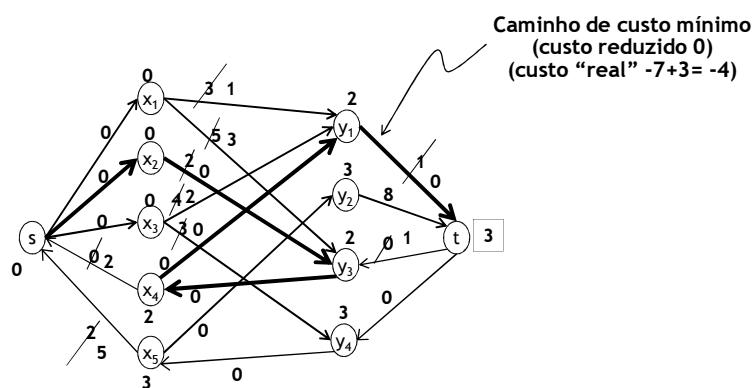
Caminho de custo mínimo
(custo reduzido 0)
(custo “real” -9)

* Resolução como problema de fluxo de custo mínimo - 2^a iteração

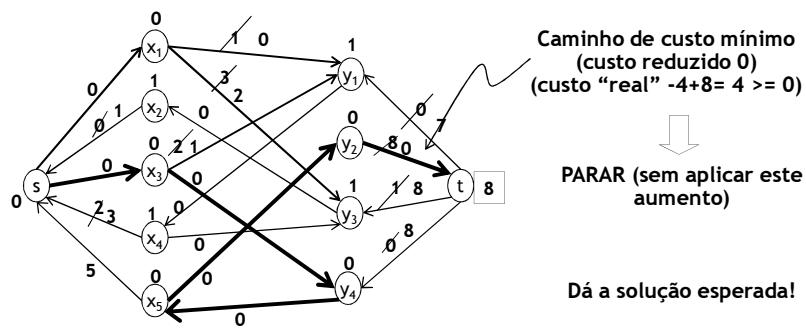


Caminho de custo mínimo
(custo reduzido 0)
(custo “real” -9+2= -7)

* Resolução como problema de fluxo de custo mínimo - 3^a iteração



* Resolução como problema de fluxo de custo mínimo - 4^a iteração

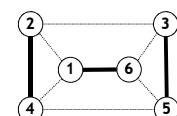
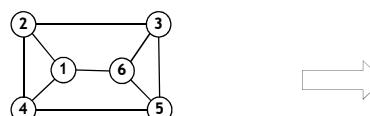


Dá a solução esperada!

* Eficiência da resolução como problema de fluxo de custo mínimo

- Neste caso o nº máximo de iterações (valor máximo de fluxo) é $|V|/2$, pois em cada iteração há mais 2 vértices que são emparelhados
- Assim, o tempo total fica $O(|E| |V| \log |V|)$ usando heap binário, podendo ser melhorado para $O(|E| |V| \log_{\lceil |E|/|V|+1 \rceil} |V|)$ usando d-heap com $d = \lceil |E|/|V|+1 \rceil$
- Se o grafo for denso ($|E| \approx |V|^2$), fica $O(|V|^3)$, que é tão bom como o algoritmo Húngaro
- Se o grafo for esparsão ($|E| \approx |V|$), fica $(O(|V|^2 \log |V|))$, que é melhor que o algoritmo Húngaro

3. Emparelhamento de tamanho máximo num grafo genérico



Quantas soluções há?

Generalização do método dos caminhos de aumento para grafos genéricos

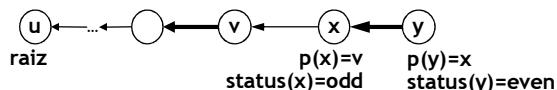
- Começa-se por descrever um algoritmo para encontrar um caminho de aumento num grafo bipartido $G=(V, E)$ em relação a um emparelhamento M , que é generalizável posteriormente para o caso de grafos genéricos, contrariamente aos algoritmos baseados em fluxo em redes

* Algoritmo para encontrar um caminho de aumento num grafo bipartido (1/2)

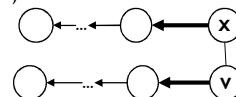
- Ideia geral: Construir uma floresta de visita do grafo ao longo de “caminhos alternados” (formados alternadamente por arestas livres.emparelhadas) partindo dos vértices livres (cada vértice livre é raiz dumha árvore), até encontrar uma aresta que liga duas árvores formando um caminho de aumento válido de raiz a raiz
- Considera-se o emparelhamento definido da seguinte forma:
 - $\text{mate}(v)$ - indica para cada vértice o seu par (null se não estiver emparelhado)
- Associa-se a cada vértice v um estado de visita $\text{status}(v)$
 - Estados possíveis:
 - unreach - não alcançado
 - even (+) - alcançado por caminho de comprimento par desde a raiz da árvore
 - odd (-) - alcançado por caminho de comprimento ímpar desde a raiz da árvore
 - Inicialmente consideram-se todos os vértices livres no estado even (cada um é raiz de uma árvore) e todos os vértices emparelhados no estado unreach
- Associa-se a cada vértice v um apontador $p(v)$ para o vértice anterior na árvore de visita

* Algoritmo para encontrar um caminho de aumento num grafo bipartido (2/2)

- Mantém-se uma fila q dos vértices pares ainda não analisadas, que é inicializada com os vértices livres
 - Em cada passo, extrai-se um vértice v da fila q e, para cada vértice adjacente x , procede-se da seguinte forma:
 - Se $status(x)=unreached$ (o que implica que está emparelhado com um vértice que se encontra também no estado *unreached*), acrescentar esse vértice e o seu par ao caminho corrente, e acrescentar o seu par à fila de vértices a analisar



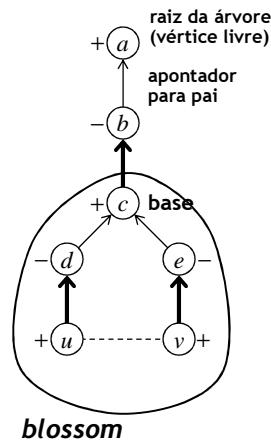
- Se $\text{status}(x)=\text{even}$ (como o grafo é bipartido, implica que v e x fazem parte de árvores diferentes), está encontrado um caminho de aumento entre as raízes das duas árvores passando por esta aresta (v,x)



- Nos outros casos, ignorar a aresta

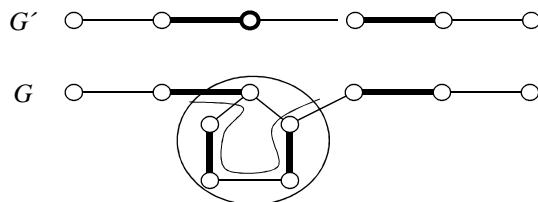
* *Blossoms*

- O algoritmo anterior falha se for encontrada uma aresta $\{u, v\}$ que liga dois vértices pares pertencentes à mesma árvore de visita
 - Não pode acontecer em grafos bipartidos, mas pode acontecer em grafos genéricos
 - O ciclo (de tamanho necessariamente ímpar) formado pelo caminho entre u e v na árvore, mais a aresta $\{u, v\}$ é chamado um *blossom* (flor, rebento)
 - O antecessor comum mais próximo de u e v na árvore (necessariamente par) é chamado a **base** do blossom
 - Pode ser a raiz da árvore de visita
 - Pode ser um dos vértices u ou v



* Método do encolhimento de *blossoms* (Edmonds, 1965)

- O problema anterior é tratado *encolhendo o blossom*, isto é, condensando todos os vértices do *blossom* num único vértice, e prosseguindo a procura de um caminho de aumento no grafo condensado (podendo ser necessário encolher novos *blossoms*)
- A correcção do método é justificada pelo seguinte teorema:
- Teorema: Seja G' o grafo formado a partir de G encolhendo o *blossom* b ; então G' contém um caminho de aumento sse G contém um caminho de aumento.



* Algoritmo de Edmonds-Karp (1/5)

- Representa-se implicitamente o conjunto V' de vértices do grafo condensado corrente $G'=(V',E')$ numa estrutura de “conjuntos disjuntos”
 - Cada conjunto representa um vértice de G' e contém todos os vértices do grafo original $G=(V,E)$ que foram condensados para esse vértice
 - Inicialmente, cada conjunto só tem um vértice original
 - Quando se encontra um blossom, faz-se a união dos seus constituintes (vértices ou blossoms)
- Mantém-se informação adicional para saber qual é a base (denotada por v') do maior blossom a que cada vértice v pertence correntemente; no caso de um vértice isolada, considera-se que a base é o próprio vértice
 - A operação $find(v)$ dos “conjuntos disjuntos” dá o maior blossom a que v pertence correntemente, representado por um vértice do mesmo, mas não temos controlo sobre qual é esse vértice, e pode mudar cada vez que se faz uma união
 - Assim, prevemos em cada vértice v um campo adicional “base”, denotado por $base(v)$, o qual só é mantido actualizado nos vértices retornados por $find()$
 - Inicialmente, faz-se $base(v)=v$ para todos os vértices
 - Cada vez que, ao explorar uma aresta (v,w) , se descobre um novo blossom com base u , aplica-se a união e faz-se depois $base(find(x)) = u$, em que x é qualquer elemento do novo blossom (pode ser u , v ou w)
 - Assim, $base(find(v))$ dá sempre a informação pretendida (v')

* Algoritmo de Edmonds-Karp (2/5)

- Mantém-se, como no caso bipartido:
 - q - fila com os vértices pares por explorar, inicializada com os vértices livres
 - $p(v)$ - antecessor de cada vértice v na floresta de visita, inicializado com null
 - $\text{status}(v)$ - estado de cada vértice v - unreached, odd ou even (inicialmente even para os vértices livres e unreached para os restantes)
- Adicionalmente, mantém-se em $\text{root}(v)$ a raiz da árvore a que cada vértice alcançado pertence correntemente
 - Inicialmente é igual ao próprio vértice nos vértices livres
- Em cada passo, retira-se o próximo vértice v da fila q e examina-se cada uma das arestas (v,w) nele incidentes, procedendo conforme explicado no slide seguinte (nos casos não mencionados não se faz nada)
- Termina com sucesso quando se encontra um caminho de aumento, ou sem sucesso quando a fila q fica vazia

* Algoritmo de Edmonds-Karp (3/5) Análise de uma aresta (v,w)

- Se $\text{status}(w) = \text{unreached}$, prolongar o caminho
 - Prolongar o caminho de visita corrente por w e $\text{mate}(w)$, fazendo
 $\text{root}(w) \leftarrow \text{root}(v)$, $\text{status}(w) \leftarrow \text{odd}$, $p(w) \leftarrow v$,
 $\text{root}(\text{mate}(w)) \leftarrow \text{root}(v)$, $\text{status}(\text{mate}(w)) \leftarrow \text{even}$, $p(\text{mate}(w)) \leftarrow w$
 - Acrescentar $\text{mate}(w)$ à fila q
- Se $\text{status}(w') = \text{even} \wedge \text{root}(v) \neq \text{root}(w)$, terminar
 - O caminho de aumento é $\text{reverse}(\text{path}(v, \text{root}(v))) \& \text{path}(w, \text{root}(w))$
(& representa concatenação; path é definido recursivamente adiante)
- Se $\text{status}(w') = \text{even} \wedge \text{root}(v) = \text{root}(w) \wedge v' \neq w'$, há um novo blossom
 - Procurar o antecessor comum mais próximo de v e w (base do novo blossom):
 $u \leftarrow \text{nca}(v, w)$ (nca é definido recursivamente adiante)
 - Formar o novo blossom (fazer o mesmo a partir de w , trocando $v \leftrightarrow w$):
 $\text{for } (x' \leftarrow \text{base}(\text{find}(v)); x' \neq u; x' \leftarrow \text{base}(\text{find}(p(x'))))$
 $\quad \text{union}(\text{find}(u), \text{find}(x'));$ {operação de conjuntos disjuntos}
 $\quad \text{if } (\text{status}(x') = \text{odd})$
 $\quad \quad q \leftarrow q \cup \{x'\};$ {pois x' pode agora funcionar como vértice par}
 $\quad \quad \text{bridge}(x') \leftarrow (v, w);$ {usado a reconstruir o caminho de aumento}
 - Registar a base do novo blossom, isto é, fazer $\text{base}(\text{find}(u)) \leftarrow u$

* Algoritmo de Edmonds-Karp (4/5)

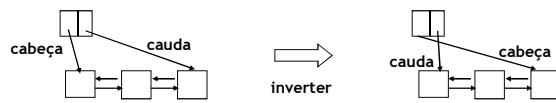
Definição recursiva de nca(x,y)

- Caminha-se no sentido da raiz da árvore em paralelo a partir dos dois vértices (v e w), até encontrar por um caminho um vértice já visitado pelo outro caminho
- Usa-se uma flag auxiliar *visited* para marcar os vértices já visitados, a qual é limpa no caminho de regresso para os vértices de partida
- Como basta percorrer as bases dos constituintes (vértices ou blossoms) do novo blossom, passa-se sempre à base do maior blossom que contém cada vértice
- A definição recursiva é então:
 - Fazer $x' \leftarrow \text{base}(\text{find}(x))$
 - Se $\text{visited}(x') = \text{true}$ {já passou aqui pelo outro caminho}, fazer $u \leftarrow x'$
 - Senão, fazer $\text{visited}(x') \leftarrow \text{true}$,
 - Se $y \neq \text{null}$, fazer $u \leftarrow \text{nca}(y, p(x'))$ {troca args para mudar de caminho}
 - Senão, fazer $u \leftarrow \text{nca}(p(x'), y)$ { $p(x')$ nunca é null neste caso}
 - Fazer $\text{visited}(x') \leftarrow \text{false}$ {limpa a flag}
 - Retornar u
- No conjunto de todas as execuções desta rotina (a procurar um caminho de aumento), cada vértice só é percorrido duas vezes no máximo

* Algoritmo de Edmonds-Karp (5/5)

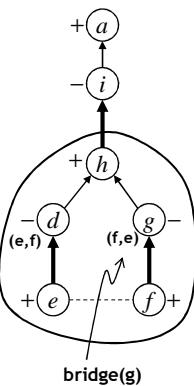
Definição recursiva de path(x,y)

- Definição recursiva de $\text{path}(x, y)$:
 - Se $x = y$, retornar $[x]$ {lista formada por um único elemento}
 - Senão, se x é par, retornar $[x, p(x)] \ \& \ \text{path}(p(p(x)), y)$
 - Senão, retornar $[x] \ \& \ \text{reverse}(\text{path}(s, \text{mate}(x))) \ \& \ \text{path}(t, y)$
em que $(s, t) = \text{bridge}(x)$
- Pode ser implementado em tempo linear ($O(|V|)$) usando uma lista reversível
 - Lista duplamente ligada com apontadores para a cabeça e cauda
 - Concatenações e inversões podem ser realizadas em tempo constante

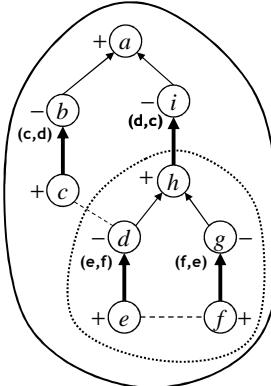


* Exemplo

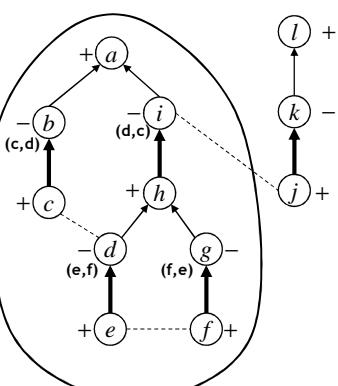
Descoberta do 1º blossom



Descoberta do 2º blossom



Descoberta de um caminho de aumento (*)



(*) a-b-c-d-e-f-g-h-i-j-k-l

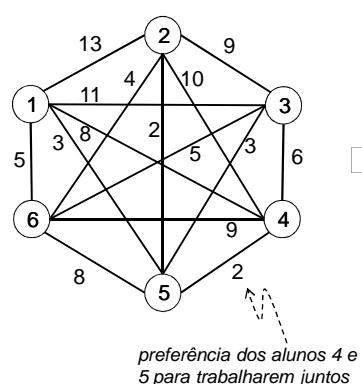
* Eficiência do algoritmo de Edmonds-Karp

- Excluindo as operações sobre a estrutura de “conjuntos disjuntos”, o tempo para descobrir um caminho de aumento é linear no tamanho do grafo, ou seja, $O(|V| + |E|)$
- No pior caso, o número de operações de busca na estrutura de conjuntos disjuntos é da ordem do número de arestas, e o número de operações de união é da ordem do número de vértices
- Prova-se que, em certos casos particulares, em que se inclui o presente caso, as operações de busca e união podem ser executadas em tempo total linear no número de operações (usando as optimizações referidas nos slides sobre “conjuntos disjuntos”)
- Assim, o tempo total para encontrar um caminho de aumento é $O(|E|)$ (supondo $|E| > |V|$), e o tempo total para encontrar um emparelhamento de tamanho máximo é $O(|E| |V|)$

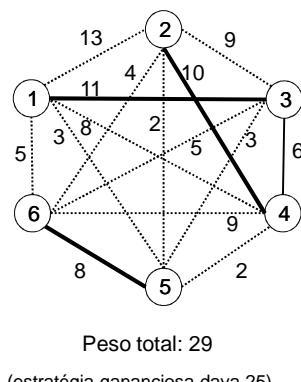
*Algoritmo de Micali-Vazirani (1980)

- Modifica o algoritmo de Edmonds-Karp, por forma a conseguir encontrar numa única passagem um conjunto maximal de caminhos de aumento disjuntos, conforme acontece no algoritmo de Hopcroft-Karp para o caso bipartido
- Consegue atingir a mesma eficiência que o algoritmo de Hopcroft-Karp, ou seja, $O(\sqrt{|V| \times |E|})$
- Ver detalhes nas referências

4. Emparelhamento de peso máximo num grafo genérico



Neste exemplo o grafo é completo



Neste exemplo o emparelhamento é **perfeito** (envolve todos os vértices)

* Algoritmos para determinação de emparelhamentos de peso máximo

- (ver bibliografia)

* Problema dos casamentos estáveis *(stable marriage)*

Problema

- Supondo que cada elemento dum grupo de n homens e n mulheres ordenou todos os de sexo oposto por ordem de preferência estrita, pretende-se determinar um emparelhamento estável
- Informalmente, um emparelhamento é, neste caso, um conjunto de n casais (monogâmicos e heterossexuais)
- Um emparelhamento E diz-se instável se e só se existir um par $(h,m) \notin E$ tal que h prefere m à sua parceira em E e m também prefere h ao seu parceiro em E
- Caso contrário, diz-se estável

Algoritmo de Gale-Shapley (1962)

ALGORITMO DE GALE-SHAPLEY (1962)

Considerar inicialmente que todas as pessoas estão livres.

Enquanto houver algum homem h livre fazer:

seja m a primeira mulher na lista de h a quem este ainda não se propôs;

se m estiver livre então

emparelhar h e m (ficam noivos)

senão

se m preferir h ao seu actual noivo h' então

emparelhar h e m (ficam noivos), voltando h' a estar livre

senão

m rejeita h e assim h continua livre.

fim

Tempo de execução: $O(n^2)$

Listas de preferências incompletas

- Surgiu na colocação de internos em hospitais
- O critério de estabilidade das soluções é reformulado:

Um emparelhamento é instável se e só se existir um candidato r e um hospital h tais que h é aceitável para r e r é aceitável para h , o candidato r não ficou colocado ou prefere h ao seu actual hospital e h ficou com vagas por preencher ou h prefere r a pelo menos um dos candidatos com que ficou

Caso contrário, diz-se estável.

* Algoritmo de Gale-Shapley com listas de preferências incompletas

ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS)

```

Considerar inicialmente que todos os internos estão livres.
Considerar também que todas as vagas nos hospitais estão livres.
Enquanto existir algum interno  $r$  livre cuja lista de preferências é não vazia
    seja  $h$  o primeiro hospital na lista de  $r$ ;
    se  $h$  não tiver vagas
        seja  $r'$  o pior interno colocado provisoriamente em  $h$ ;
         $r'$  fica livre (passa a não estar colocado);
        colocar provisoriamente  $r$  em  $h$ ;
        se  $h$  ficar sem vagas então
            seja  $s$  o pior dos colocados provisoriamente em  $h$ ;
            para cada sucessor  $s'$  de  $s$  na lista de  $h$ 
                remover  $s'$  e  $h$  das respectivas listas
    fim

```

* Propriedades do algoritmo de Gale-Shapley

- O emparelhamento obtido por este algoritmo é óptimo para os internos e péssimo para os hospitais: qualquer interno fica com o melhor hospital que pode ter em qualquer emparelhamento estável e cada hospital fica com os piores internos.
- Tempo de execução é de ordem quadrática (n^o de internos * n^o de hospitais)

Problema da colocação de professores, Portugal, 2004 (1)

- Existem professores que concorrem a vagas
- Professores concorrentes são de dois tipos:
 - professores que tinham colocação, mas que pretendem mudar de posição (se não for possível, ficam na posição anterior)
 - professores que não tinham colocação, e pretendem obter uma colocação (se não for possível, ficam sem colocação)
- Cada professor indica uma lista totalmente ordenada de vagas a que concorre
- Vagas a concurso incluem as posições anteriormente ocupadas pelos professores que pretendem mudar de posição
- Os professores já estão totalmente ordenados segundo um ranking
- Neste ranking, podem aparecer intercalados professores dos dois tipos

Problema da colocação de professores, Portugal, 2004 (2)

- O resultado da colocação deve obedecer às seguintes restrições:
 - Os professores que tinham colocação anteriormente têm de ficar colocados, nem que seja na posição anterior
 - Para cada professor e para cada posição por ele preferida em relação àquela em que foi colocado (inclui todas as posições no caso de não ter sido colocado), essa posição tem de estar ocupada por um professor com melhor ranking ou pelo professor que aí estava anteriormente colocado
- Pode ser formulado como problema de casamentos estáveis com listas de preferências incompletas
 - Internos correspondem aos professores
 - Hospitais correspondem às vagas
 - Cada vaga prefere 1º o professor que aí estava colocado anteriormente, e depois todos os outros pela ordem do ranking

Referências e mais informação

- “Efficient Algorithms for Finding Efficient Maximum Matchings in Graphs”, Zvi Galil, ACM Computing Surveys, March 1986
- “Emparelhamentos, Casamentos Estáveis e Algoritmos de Colocação de Professores”, Ana Paula Tomás, Faculdade de Ciências da Universidade do Porto, Technical Report Series: DCC-05-02, Março de 2005
- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
- “The General Maximum Matching Algorithm of Micali and Vazirani”, Paul A. Peterson, Michael C. Loui, Algoritmica (1988) 3: 511: 533, Springer-Verlag

Estruturas

Conjuntos Disjuntos

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
FEUP, MIEIC, CPAL, 2010/2011

Conjuntos Disjuntos

Objectivo

- ° resolver eficientemente o problema da equivalência
- ° estrutura de dados simples (vector)
- ° implementação rápida

Desempenho

- ° análise complicada de avaliar para implementações mais complexas

Uso

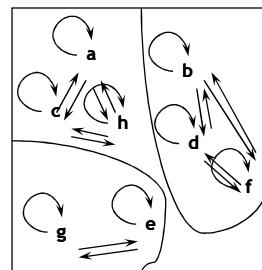
- ° problemas de grafos
- ° equivalência de tipos em compiladores

Conjuntos Disjuntos

- É uma colecção $C=\{S_1, S_2, \dots, S_k\}$ de conjuntos dinâmicos distintos
- Cada conjunto é identificado por um dos seus elementos, chamado *representativo*
- Operações básicas sobre os conjuntos incluem:
 - **MAKE-SET(x)**: cria um novo conjunto com o elemento x apenas. Assume-se que x não existe noutro conjunto de C .
 - **UNION(x,y)**: combina dois conjuntos contendo x e y em um novo conj. único. Um novo representativo é seleccionado.
 - **FIND-SET(x)**: retorna o representativo do conjunto contendo x .

Relações de equivalência

- relação R (ou \sim) definida num conjunto S se
 - $a R b =$ Verdadeiro ou $a R b =$ Falso $\forall a, b \in S$
 - $a R b$ ou $a \sim b$
 $\Rightarrow a$ está relacionado com b
- propriedades das relações de equivalência
 - **reflexiva** $a R a, \forall a \in S$
 - **simétrica** $a R b \rightarrow b R a$
 - **transitiva** $a R b, b R c \rightarrow a R c$



Relações de equivalência

- exemplos de relações
 - \leq - reflexiva, transitiva; não é simétrica \Rightarrow não é de equivalência
 - "pertencer ao mesmo país" (S é o conjunto das cidades)
 - reflexiva, simétrica, transitiva \Rightarrow relação de equivalência
- Classe de equivalência de $a \in S$
 - subconjunto de S que contém os elementos relacionados com a
 - relação de equivalência induz uma partição de S : cada elemento pertence exactamente a uma classe
- Para decidir se $a \sim b$, é necessário apenas comprovar se a e b estão na mesma classe de equivalência

Problema da equivalência dinâmica

- R : relação de equivalência
- Problema: dados a e b , determinar se $a R b$
- Solução: relação armazenada numa matriz bidimensional de booleanos
 - \Rightarrow resposta em tempo constante
- Dificuldade: relações definidas implicitamente
 - $\{a_1, a_2, a_3, a_4, a_5\}$ (25 pares)
 - $a_1 R a_2, a_3 R a_4, a_5 R a_1, a_4 R a_2 \Rightarrow$ todos relacionados
 - pretende-se obter esta conclusão rapidamente
- Observação: $a \sim b \leftarrow a$ e b pertencem à mesma classe

Algoritmo abstracto

- Entrada: colecção de n conjuntos, cada um com seu elemento
 - disjuntos
 - só propriedade reflexiva
- Duas operações
 - ° Busca - devolve o nome do conjunto que contém um dado elemento
 - ° União - dados dois conjuntos substitui-os pela respectiva união (preserva a disjunção da colecção)
- Método: acrescentar o par $a \sim b$ à relação
 - ° usa Busca em a e em b p/ verificar se pertencem já à mesma classe
 - se sim, o par é redundante
 - senão, aplica União às respectivas classes
 - ° algoritmo **dinâmico** (os conjuntos são alterados por União) e **em-linha** (cada Busca tem que ser respondida antes de o algoritmo continuar)
 - ° valores dos elementos irrelevantes \Rightarrow basta numerá-los com uma função de dispersão
 - ° nomes concretos dos conjuntos irrelevantes \Rightarrow basta que a igualdade funcione

Privilegiando a Busca *

Busca com tempo constante para o pior caso:

- ° implementação: vector indexado pelos elementos indica nome da classe respectiva
- ° Busca fica uma consulta de $O(1)$
- ° União(a, b): se Busca(a) = i e Busca(b) = j , pode-se percorrer o vector mudando todos os i 's para $j \Rightarrow$ custo $\Theta(n)$
- ° para $n-1$ Uniões (o máximo até ter tudo numa só classe) \Rightarrow tempo $\Theta(n^2)$
- ° se houver $\Omega(n^2)$ Busca, o tempo é $O(1)$ para operação elementar; não é mau

Melhoramentos:

- ° colocar os elementos da mesma classe numa lista ligada para saltar directamente de uns para os outros ao fazer a alteração do nome da classe (mantém o tempo do pior caso em $O(n^2)$)
- ° registar o tamanho da classe de equivalência para alterar sempre a mais pequena; cada elemento é alterado no máximo $\log n$ vezes (cada fusão duplica a classe) \Rightarrow com $n-1$ fusões e m Buscas $O(n \log n + m)$

Privilegiando a União *

Representar cada conjunto como uma árvore

- a raiz serve como nome do conjunto
- as árvores podem não ser binárias: cada nó só tem um apontador para o pai
- as árvores são armazenadas implicitamente num vector
 - $p[i]$ contém o número do pai do elemento i
 - se i for uma raiz $p[i] = 0$

União: fusão de duas árvores

- Pôr a raiz de uma a apontar para a outra ($O(1)$)
- Convenção: União(x, y) tem como raiz x

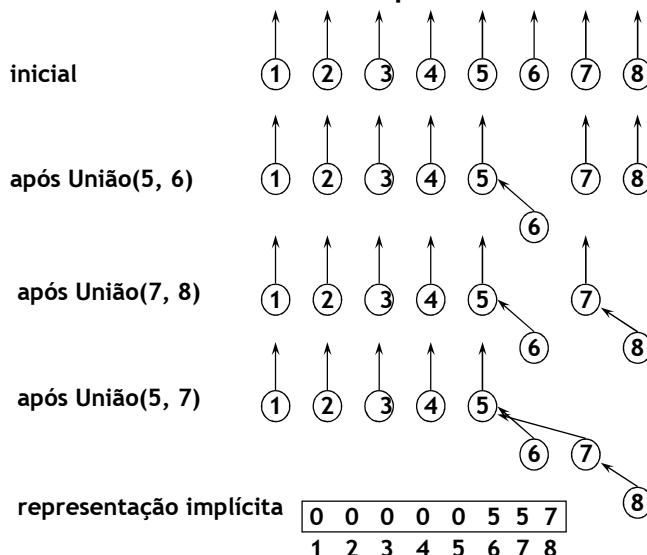
Busca(x) devolve a raiz da árvore que contém x

- tempo proporcional à profundidade de x ($n-1$ no pior caso)
- m operações podem durar $O(mn)$ no pior caso

Não é possível ter tempo constante simultaneamente para União e Busca



Exemplo



Implementação

construtor

```
Disj_Sets(int Num_Elements)
{
    S_Size = Num_Elements;
    S_Array = new int [S_Size + 1];
    for( int i=0; i <= S_Size; i++)
        S_Array[i] = 0;
}
```

União (fraco)

```
Void
Unir( int Root1, int Root2 )
{
    S_Array [ Root2 ] = Root1;
}
```

Busca simples

```
int
Busca( int X )
{
    if ( S_Array [ X ] <= 0 )
        return X;
    else
        return Busca( S_Array[ X ] );
}
```

Análise no caso médio *

Como definir "médio" relativamente à operação União?

- depende do modelo escolhido (ver exemplo anterior; última situação)
- 1 como no exemplo restaram 5 árvores, há $5 \times 4 = 20$ resultados equiprováveis da próxima União
 - 2/5 de hipóteses de envolver a árvore maior
 - 2 considerando como equiprováveis as Uniões entre dois quaisquer elementos de árvores diferentes
 - há 6 maneiras de fundir dois elementos de {1, 2, 3, 4} e 16 maneiras de fundir um elemento de {1, 2, 3, 4} e um de {5, 6, 7, 8}
 - probabilidade de a árvore maior estar envolvida: 16/22

O tempo médio depende do modelo: $\Theta(m)$, $\Theta(m \log n)$, $\Theta(m n)$

- tempo quadrático é mau, mas evitável

União melhorada *

após União(4, 5)
(altura 3)

```

graph TD
    1((1)) --> 2((2))
    2((2)) --> 3((3))
    3((3)) --> 4((4))
    4((4)) --> 5((5))
    5((5)) --- 6((6))
    5((5)) --- 7((7))
    5((5)) --- 8((8))
  
```

União-por-Tamanho

colocar a árvore menor como sub-árvore da maior (arbitrar em caso de empate)

após União(4, 5)
(altura 2)

```

graph TD
    1((1)) --> 2((2))
    2((2)) --> 3((3))
    3((3)) --> 4((4))
    4((4)) --- 5((5))
    5((5)) --- 6((6))
    5((5)) --- 7((7))
    5((5)) --- 8((8))
  
```

FEUP Universidade do Porto
Faculdade de Engenharia
Conjuntos Disjuntos - CAL, 2010/11
13

União-por-Tamanho *

profundidade de cada nó – nunca superior a $\log n$

- um nó começa por ter profundidade 0
- cada aumento de profundidade resulta de uma união que produz uma árvore pelo menos com o dobro do tamanho
- logo, há no máximo $\log n$ aumentos de profundidade
- Busca é $O(\log n)$; m operações é $O(m \log n)$

Pior caso para $n=16$ (União entre árvores de igual tamanho)

```

graph TD
    1((1)) --- 2((2))
    1((1)) --- 3((3))
    1((1)) --- 5((5))
    1((1)) --- 9((9))
    2((2)) --- 4((4))
    2((2)) --- 6((6))
    2((2)) --- 7((7))
    3((3)) --- 8((8))
    5((5)) --- 10((10))
    5((5)) --- 11((11))
    5((5)) --- 13((13))
    9((9)) --- 12((12))
    9((9)) --- 14((14))
    9((9)) --- 15((15))
    9((9)) --- 16((16))
  
```

registar a dimensão de cada árvore (na raiz respectiva e com sinal negativo)

- o resultado de uma União tem dimensão igual à soma das duas anteriores
- para m operações, dá $O(m)$

FEUP Universidade do Porto
Faculdade de Engenharia
Conjuntos Disjuntos - CAL, 2010/11
14

União-por-Altura *

- em vez da dimensão, regista-se a altura
 - coloca-se a árvore mais baixa como subárvore da mais alta
 - altura só se altera quando as árvores a fundir têm a mesma altura
- representação vectorial da situação após União(4, 5)

União-por-Tamanho

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

União-por-Altura

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

União (melhorado)

```
void
União_por_Altura( int Root1, int Root2 )
{
    if (S_Array [ Root2 ] < S_Array[ Root1 ] )
        S_Array[ Root1 ] = Root2;
    else
    {
        if (S_Array [ Root1 ] == S_Array[ Root2 ] )
            S_Array [ Root1 ]--;
        S_Array [ Root2 ] = Root1;
    }
}
```



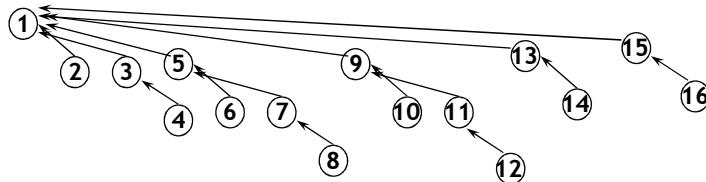
Compressão *

- algoritmo descrito é linear na maior parte das situações; mas no pior caso é $O(m \log n)$
 - já não é fácil melhorar o União : actuar na Busca

Compressão do caminho

ao executar Busca(x), todos os nós no caminho de x até à raiz ficam com a raiz como pai

Compressão após Busca_e_Compressão(15)



Busca modificada *

Busca com compressão

- profundidade de vários nós diminui
- com União arbitrária, a compressão garante m operações, no pior caso, em tempo $O(m \log n)$
- desconhece-se qual é, em média, o comportamento só da compressão
- a compressão é compatível com União-por-Tamanho
- não é completamente compatível com União-por-Altura: não é fácil computar eficientemente as alturas modificadas pela compressão
- não se modificam os valores: passam a ser entendidos como estimativas da altura, designados por nível
- ambos os métodos de União garantem m operações em tempo linear: não é evidente que a compressão traga vantagem em tempo médio: melhora o tempo no pior caso; a análise é complexa, apesar da simplicidade do algoritmo

```
int Busca ( int X )
{
    if ( S_Array [ X ] <= 0 )
        return X;
    else
        return S_Array[ X ] = Busca (
            S_Array[ X ] );
}
```

Aplicação *

- rede de computadores com uma lista de ligações bidireccionais; cada ligação permite a transferência de ficheiros de um computador para o outro
 - é possível enviar um ficheiro de um qualquer nó da rede para qualquer outro?
 - o problema deve ser resolvido em-linha, com as ligações apresentadas uma de cada vez
- o algoritmo começa por pôr cada computador em seu conjunto
 - o invariante é que dois computadores podem transferir ficheiros se estiverem no mesmo conjunto
 - esta capacidade determina uma relação de equivalência
 - à medida que se lêem as ligações vão-se fundindo os conjuntos
- o grafo da capacidade de transferência é conexo se no fim houver um único conjunto
 - com m ligações e n computadores o espaço requerido é $O(n)$
 - com União-por-Tamanho e compressão de caminho obtém-se um tempo no pior caso praticamente linear

Lema (para resultados de complexidade *)

Ao executar uma sequência de Uniões, um nó de nível r tem que ter 2^r descendentes (incluindo o próprio)

Prova por indução

- a base, $r = 0$ é verdadeira: $2^0 = 1$, e uma folha só tem um descendente (o próprio nó)
- seja T uma árvore de nível r com o mínimo número de descendentes e seja x a raiz de T
- suponha-se que a última União que envolveu x foi entre T_1 e T_2 e que x era a raiz de T_1
- se T_1 tivesse nível r , T_1 seria uma árvore de altura r com menos descendentes do que T , o que contradiz o pressuposto de T ser uma árvore com o número mínimo de descendentes. Portanto, o nível de $T_1 \leq r-1$. O nível de $T_2 \leq$ nível de T_1 . Uma vez que o nível de T é r e o nível só pode aumentar devido a T_2 , segue-se que nível de $T_2 = r-1$. Então também o nível de $T_1 = r-1$.
- pela hipótese de indução, cada uma das árvores tem $2^{(r-1)}$ descendentes, dando um total de 2^r e estabelecendo o lema.



Strings

Algoritmos em Strings

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
FEUP, MIEIC, CAL, 2010/2011

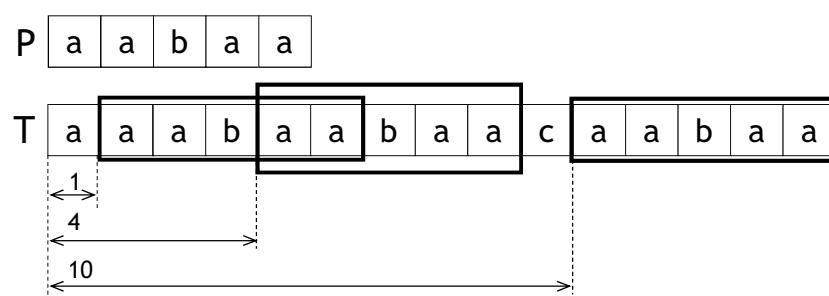
Índice

- Pesquisa exacta (*string matching*)
- Pesquisa aproximada (*approximate string matching*)
- Outros problemas

Pesquisa exacta (*string matching*)

Problema

- Encontrar todas as ocorrências de um padrão P num texto T
 - P e T são cadeias de caracteres
 - Ocorrências são definidas pela deslocação em relação ao início do texto
 - Ocorrências podem ser sobrepostas

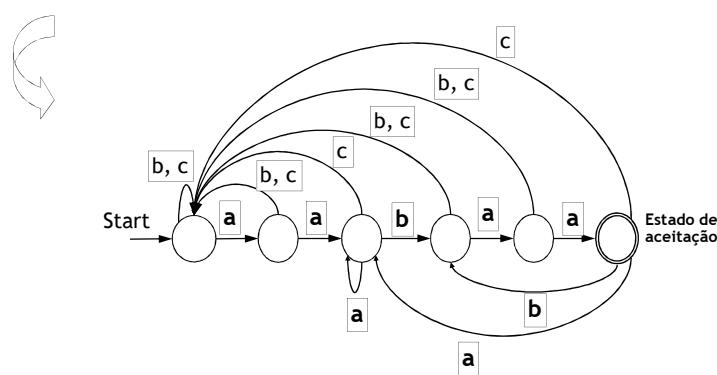


Algoritmos

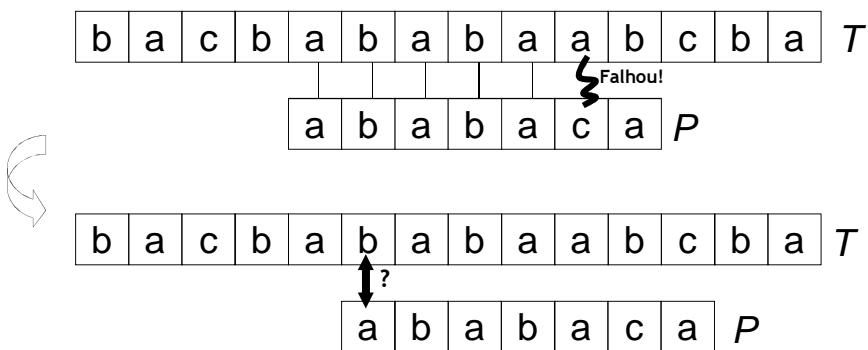
- Algoritmo *naive*
 - Para cada deslocamento possível, compara desde o início do padrão
 - Ineficiente se o padrão for comprido: $O(|P| \cdot |T|)$
- Algoritmo baseado em autómato finito
 - Pré-processamento: gerar autómato finito correspondente ao padrão
 - Permite depois analisar o texto em tempo linear $O(|T|)$, pois cada carácter só precisa de ser processado uma vez
 - Mas tempo e espaço requerido pelo pré-processamento pode ser elevado: $O(|P| \cdot |\Sigma|)$, em que $|\Sigma|$ é o tamanho do alfabeto
- Algoritmo de Knuth-Morris-Pratt
 - Consegue fazer um pré-processamento do padrão em tempo $O(|P|)$, sem chegar a gerar explicitamente um autómato, seguido de processamento do texto em $O(|T|)$, dando total $O(|T| + |P|)$

Autómato finito correspondente ao padrão

$P \quad \boxed{a \ a \ b \ a \ a} \quad \Sigma = \{a, b, c\}$

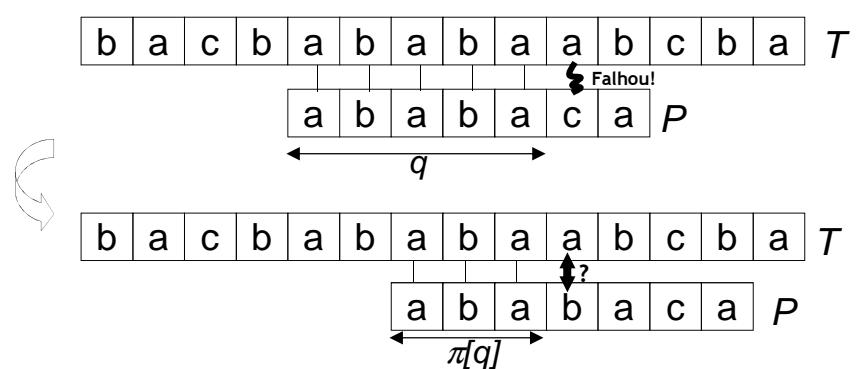


Algoritmo *naive*



Desloca-se o padrão uma casa para a direita e recomeça-se a comparação do início do padrão! Ineficiente: $O(|P| |T|)$

Algoritmo de Knuth-Morris-Pratt



Desloca-se o padrão para a direita de uma forma que permite continuar a comparação na mesma posição do texto! Evita comparações inúteis!

Deslocamento é determinado por uma função $\pi[q]$ calculada numa fase de pré-processamento do padrão!

Pré-processamento do padrão

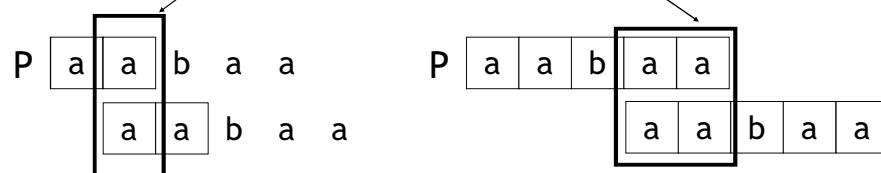
- Compara-se o padrão com deslocações do mesmo, para determinar a **função prefixo**

$$\pi[q] = \max \{k: 0 \leq k < q \text{ e } P[1..k] = P[(q-k+1)..q]\}$$

- $q = 1, \dots, |P|$
- $P[i..j]$ - substring entre índices i e j
- Índices a começar em 1
- $\pi[q]$ é o comprimento do maior prefixo de P que é um sufixo próprio do prefixo de P de comprimento q

Pré-processamento do padrão

q	1	2	3	4	5
$P[q]$	a	a	b	a	a
$\pi[q]$	0	1	0	1	2



Pseudo-código

KMP-MATCHER(T, P)

```

1   $n \leftarrow \text{length}[T]$ 
2   $m \leftarrow \text{length}[P]$ 
3   $\pi \leftarrow \text{COMPUTE-PREFIX-FUNCTION}(P)$ 
4   $q \leftarrow 0$                                  $\triangleright$  Number of characters matched.
5  for  $i \leftarrow 1$  to  $n$                    $\triangleright$  Scan the text from left to right.
6    do while  $q > 0$  and  $P[q + 1] \neq T[i]$ 
7      do  $q \leftarrow \pi[q]$            $\triangleright$  Next character does not match.
8      if  $P[q + 1] = T[i]$ 
9        then  $q \leftarrow q + 1$            $\triangleright$  Next character matches.
10     if  $q = m$                        $\triangleright$  Is all of  $P$  matched?
11       then print “Pattern occurs with shift”  $i - m$ 
12        $q \leftarrow \pi[q]$            $\triangleright$  Look for the next match.
```

Pseudo-código

COMPUTE-PREFIX-FUNCTION(P)

```

1   $m \leftarrow \text{length}[P]$ 
2   $\pi[1] \leftarrow 0$ 
3   $k \leftarrow 0$ 
4  for  $q \leftarrow 2$  to  $m$ 
5    do while  $k > 0$  and  $P[k + 1] \neq P[q]$ 
6      do  $k \leftarrow \pi[k]$ 
7      if  $P[k + 1] = P[q]$ 
8        then  $k \leftarrow k + 1$ 
9       $\pi[q] \leftarrow k$ 
10   return  $\pi$ 
```

Eficiência do algoritmo de Knuth-Morris-Pratt

- KMP-MATCHER (sem incluir COMPUTE-PREFIX-FUNCTION)
 - Eficiência depende do nº de iterações do ciclo “while” interno
 - Dado que $0 \leq \pi[q] < q$, cada vez que a instrução 7 é executada, o valor de q é decrementado de pelo menos 1, sem nunca chegar a ser negativo
 - Dado que o valor de q começa em 0 e só é incrementado no máximo n vezes (+1 de cada vez, na linha 9), o nº máximo de vezes que pode ser decrementado (nas linhas 7 e 12) é também n
 - ⇒ Nº máximo de iterações do ciclo “while” interno (no conjunto de todas as iterações do ciclo “for” externo) é n
 - ⇒ Tempo de execução da rotina é $O(n)$, i.e., $O(|T|)$
- COMPUTE-PREFIX-FUNCTION
 - Seguindo o mesmo raciocínio, tempo de execução é $O(m)$, i.e., $O(|P|)$
- Total: $O(n+m)$, isto é, $O(|T| + |P|)$

Pesquisa aproximada (*approximate string matching*)

Problema

misplaced ?
misplaced
mislead

INPUT

OUTPUT

Input description: A text string T and a pattern string P . An edit cost bound k .

Problem description: Can we transform T to P using at most k insertions, deletions, and substitutions?
(Ou: qual é o grau de semelhança entre P e T ?)

Distância de edição entre duas strings

- A *distância de edição* entre P (*pattern string*) e T (*text string*) é o menor número de alterações necessárias para transformar T em P , em que as alterações podem ser:
 - substituir um carácter por outro
 - inserir um carácter
 - eliminar um carácter

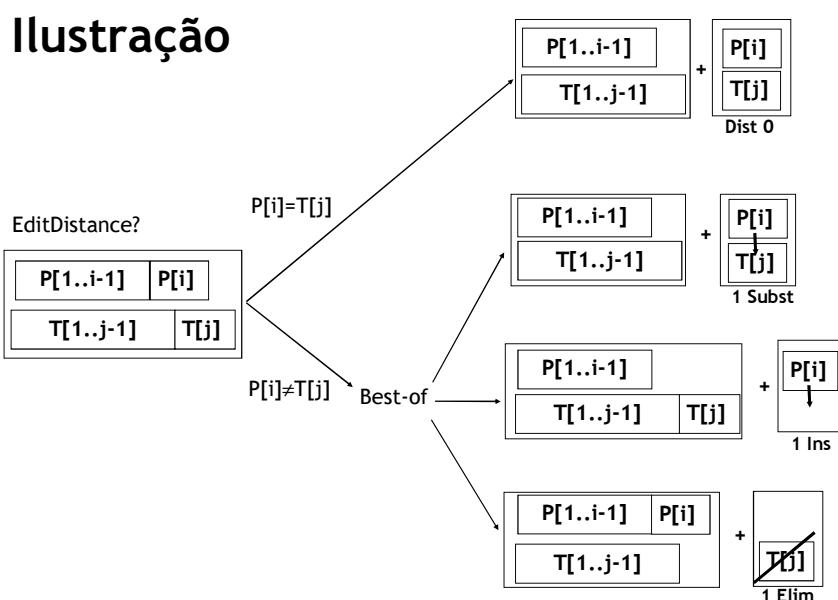
$P: \text{abcdefghijkl}$
 ||||| |||| |
 $T: \text{bcdeffghixkl}$
 ↑ ↑ ↑
 inserção eliminação substituição

EditDistance(P, T)=3

Formulação recursiva

- $D[i,j] = \text{EditDistance}(P[1..i], T[1..j]), 0 \leq i \leq |P|, 0 \leq j \leq |T|$
- Caso recursivo ($i > 0$ e $j > 0$):
 - Se $P[i] = T[j]$, então $D[i, j] = D[i-1, j-1]$
 - Senão, escolhe-se a operação de edição que sai mais barata; isto é, $D[i,j]$ é o mínimo de:
 - $1 + D[i-1, j-1]$ (substituição de $T[j]$ por $P[i]$)
 - $1 + D[i-1, j]$ (inserção de $P[i]$ a seguir a $T[j]$)
 - $1 + D[i, j-1]$ (eliminação de $T[j]$)
- Condições fronteira:
 - $D[0, j] = j, D[i, 0] = i$ (porquê?)

Ilustração



Matriz de programação dinâmica

		T												
		D[i,j]												
		b	c	d	e	f	f	g	h	i	x	k	l	
		0	1	2	3	4	5	6	7	8	9	10	11	12
P	a	1	2	3	4	5	6	7	8	9	10	11	12	
	b	2	1	2	3	4	5	6	7	8	9	10	11	12
	c	3	2	1	2	3	4	5	6	7	8	9	10	11
	d	4	3	2	1	2	3	4	5	6	7	8	9	10
	e	5	4	3	2	1	2	3	4	5	6	7	8	9
	f	6	5	4	3	2	1	2	3	4	5	6	7	8
	g	7	6	5	4	3	2	2	2	3	4	5	6	7
	h	8	7	6	5	4	3	3	3	2	3	4	5	6
	i	9	8	7	6	5	4	4	3	2	3	4	5	5
	j	10	9	8	7	6	5	5	4	3	3	4	5	5
	k	11	10	9	8	7	6	6	5	4	4	3	4	4
	l	12	11	10	9	8	7	7	7	6	5	5	4	3

Pseudo-código

Tempo e espaço: $O(|P| \cdot |T|)$

```

EditDistance(P,T) {
    // inicialização
    for i = 0 to |P| do D[i,0] = i
    for j = 0 to |T| do D[0,j] = j
    // recorrência
    for i = 1 to |P| do
        for j = 1 to |T| do
            if P[i] == T[j] then D[i,j] = D[i-1,j-1]
            else D[i,j] = 1 + min(D[i-1,j-1],
                                   D[i-1,j],
                                   D[i,j-1])
    // finalização
    return D[|P|, |T|]
}

```

Optimização de espaço

Espaço: $O(|T|)$

```

EditDistance(P,T) {
    // inicialização
    for j = 0 to |T| do D[j] = j  // D[0,j]
    // recorrência
    for i = 1 to |P| do
        old = D[0] // guarda D[i-1,0]
        D[0] = i   // inicializa D[i, 0]
        for j = 1 to |T| do
            if P[i] == T[j] then new = old
            else new = 1 + min(old,
                Ainda tem valor anterior D[i-1,j] → D[j],
                D[j-1])
            old = D[j]
            D[j] = new
    // finalização
    return D[|T|]
}

```

Já tem valor da iteração corrente, i.e., $D[i, j-1]$

Outros problemas

- Sub-sequência comum mais comprida (*longest common subsequence*)
 - Formada por caracteres não necessariamente consecutivos
 - ABD ? ABCDEF (delete)
- Substring comum mais comprida (*longest common substring*)
 - Formada por caracteres consecutivos
 - ABAB (BAB) BABA (BA) ABBA -> {AB, BA} (tamnho 2)
- Compressão de texto com códigos de Huffman
- Criptografia

Referências e mais informação

- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
 - Fonte consultada para o “matching” exato
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
 - Fonte consultada para o “matching” aproximado
 - Discute como se use o cálculo da distância de edição para encontrar num texto T a substring que faz o melhor “match” com um padrão de pesquisa P
- *Com base em slides de J. P. Faria*

Ficheiros

Algoritmos em Strings (compressão de texto)

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

FEUP, MIEIC, CAL, 2010/2011

Teoria da Informação

- O que é?
 - “É uma ferramenta matemática para determinar a quantidade mínima de dados para representar informação”
- Representação da Informação
 - Como é que se representa texto?
 - Como é que se representam imagens?
 - Como é que se representa som?
 - Técnicas simples de correcção de erros?
 - Dispositivos de armazenamento de informação?

Teoria da Informação

■ Por que comprimir?

- Preencher o hiato entre procura e capacidade
- Utilizadores têm procurado aplicações com médias cada vez mais sofisticados (Web 2.0)
- Meios de transmissão e armazenamento são limitados

Por exemplo:

Livro de 800 páginas; cada página com 40 linhas; cada linha com 80 caracteres:
 $800 * 40 * 80$ (* 1 byte por carácter) = 2,44 MB

Vídeo digital c/ “qualidade de TV digital” (aproximadamente):

1 segundo ~ 216Mbits

2 horas ~ 194GB = 42 DVDs (ou 304 CD-ROMs)!

- “compressão vai se tornar redundante em breve, com as capacidades de armazenamento e transmissão a aumentarem” ... (Será?!!!)

Técnicas de compressão

■ Codificador fonte e descodificador destino

- Em sistemas multimédia, a informação é frequentemente comprimida antes de ser armazenada ou transmitida

Algoritmos de compressão: principal tarefa do codificador fonte

Algoritmos de descompressão: principal tarefa do descodificador destino

- Implementação dos algoritmos de compressão/descodificação

Em Software: quando o tempo para compressão/descompressão não é crítico

Em Hardware: quando a aplicação é dependente do tempo, ou seja, quando o tempo para compressão/descompressão é crítico

Representação de caracteres

- ASCII: *American Standard Code for Information Interchange*
- Tradicionalmente utilizava-se 7bits para representar os diversos caracteres
 - 7bits → 128 combinações diferentes possíveis
 - Por exemplo: ‘A’ = $(1000001)_2 = (65)_{10}$
- Mais tarde, os 7bits foram estendidos a 8, permitindo assim representar 256 caracteres diferentes
 - Unicode (16bits) → 65.536
 - ISO* (36bits) → 17M

**International Organization for Standardization*

Representação de caracteres

- ASCII: *American Standard Code for Information Interchange*
- Tradicionalmente utilizava-se 7bits para representar os diversos caracteres
 - 7bits → 128 combinações diferentes possíveis
 - Por exemplo: ‘A’ = $(1000001)_2 = (65)_{10}$
- Mais tarde, os 7bits foram estendidos a 8, permitindo assim representar 256 caracteres diferentes
 - Unicode (16bits) → 65.536
 - ISO* (36bits) → 17M

**International Organization for Standardization*

Representação de caracteres

- Tabela ASCII (7bits)

Left Digit Digit(s)	ASCII									
	0	1	2	3	4	5	6	7	8	9
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT
1	LF	VT	FF	CR	SO	SI	DLE	DC1	DC2	DC3
2	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS
3	RS	US	□	!	“	#	\$	%	&	'
4	()	*	+	,	-	.	/	0	1
5	2	3	4	5	6	7	8	9	:	;
6	<	=	>	?	@	A	B	C	D	E
7	F	G	H	I	J	K	L	M	N	O
8	P	Q	R	S	T	U	V	W	X	Y
9	Z	[\]	^	-	`	a	b	c
10	d	e	f	g	h	i	j	k	l	m
11	n	o	p	q	r	s	t	u	v	w
12	x	y	z	{		}	~	DEL		

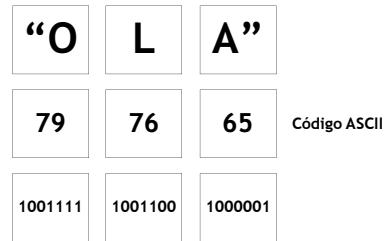
Representação de caracteres

- Unicode (16bits)

Code (Hex)	Character	Source
0041	A	English (Latin)
042F	Я	
OE09	҆	Thai
13EA	ጀ	Cherokee
211E	Rx	Letterlike Symbols
21CC	⇒	Arrows
282F	⠼	Braille
345F	ߵ	Chinese/Japanese/ Korean (Common)

Representação de textos

- A representação de texto é simplesmente uma sequência de caracteres



Técnicas de compressão de texto

- Apesar do espaço de armazenamento estar continuamente a aumentar, é desejável por vezes comprimir dados
 - Transmissão pela rede
 - Armazenamento de longa duração
 - Em geral... maior eficiência e aproveitamento de recursos
- Três métodos comuns de compressão de texto
 - *Keyword encoding*
 - *Run-length encoding (RLE)*
 - *Huffman codes*

Keyword encoding

- Substituir palavras muito comuns por caracteres especiais ou sequências especiais de caracteres
- As palavras são substituídas de acordo com uma tabela de frequências (ocorrências)

Chave	Significado
%	Carro
\$	Acidente
&	Senhor
#	Do

Keyword encoding (exemplo)

- “No acidente estiveram envolvidos três carros. O carro do senhor António ficou destruído. O carro do senhor José não sofreu grandes danos no acidente. O carro do senhor Carlos... bom, depois do acidente, nem se pode chamar aquilo um carro...”
→ 241bytes
- “No \$ estiveram envolvidos três carros. O % # & António ficou destruído. O % # & José não sofreu grandes danos no \$. O % # & Carlos... bom, depois # \$, nem se pode chamar aquilo um %...”
→ 185bytes (76%)

Run-length encoding (RLE)

- Tipicamente utilizado quando o mesmo padrão/letra surge muitas vezes seguidas numa sequência de dados;
- Não é comum em texto, mas em muitos outros tipos de dados (por exemplo: imagem, vídeo)
- Técnica utilizada em muitas aplicações comuns. Basicamente, uma sequência de caracteres que se repetem é substituída por:
 - um marcador especial (*)
 - o carácter em questão
 - número vezes que o carácter aparece

AAAAAAAAAAA → *A10

AABBBBBBBBAMMKKKKKKKKKM → AA*B8AMM*K9M

Algoritmo de Huffman

Codificação constante

- Código de tamanho fixo.

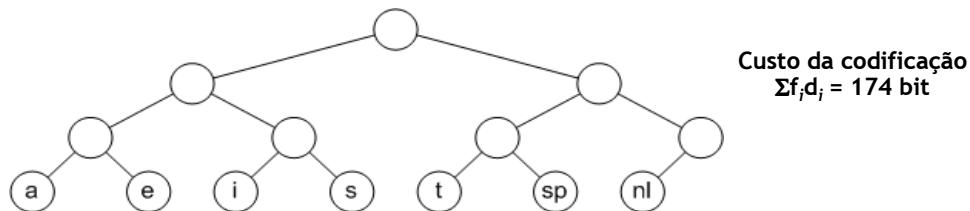
- Se $|\text{alfabeto}| = C \rightarrow$ código com $\lceil \log_2(C) \rceil$ bit
- Ex: caracteres ASCII visíveis $\cong 100 \rightarrow$ necessário código de 7 bit

- Representação possível

- Árvore binária com caracteres só nas folhas
- Na descodificação:
 - se é folha, então encontrou-se o carácter
 - se o bit corrente do código for 0, visita-se a sub-árvore esquerda
 - se o bit corrente do código for 1, visita-se a sub-árvore direita

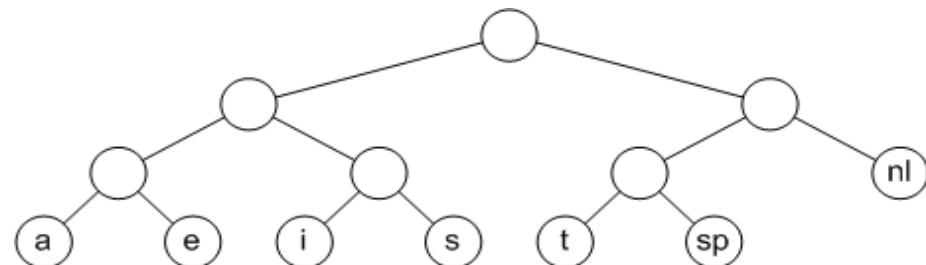
Codificação constante (exemplo)

Carácter	Código	Frequência	Total bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
espaço	101	13	39
newline	110	1	3



Codificação constante (cont)

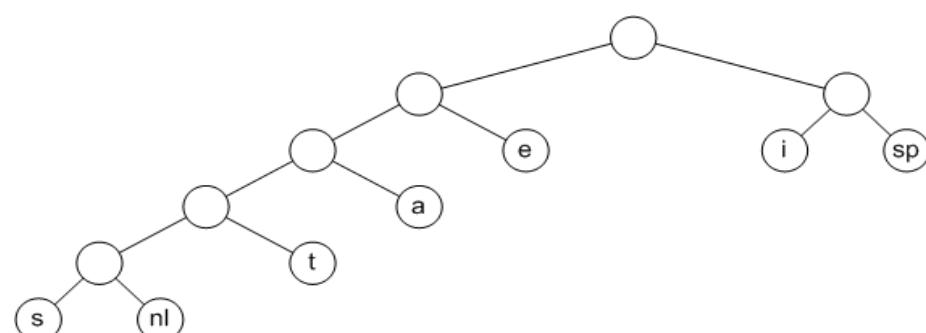
- Reduzindo o custo de codificação...



Custo da codificação $\sum f_i d_i = 173$ bit

Codificação constante (cont)

- Codificação óptima...



Custo da codificação $\sum f_i d_i = 146$ bit

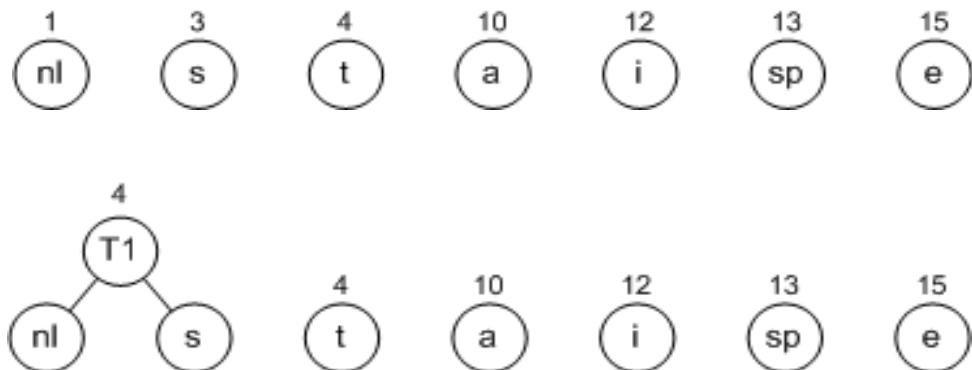
Códigos de Huffman

- Código de tamanho variável
Caracteres mais frequentes → código mais pequeno
- Utiliza uma árvore binária com os símbolos só nas folhas
- Os símbolos nas folhas permitem descodificação não ambígua
(código não prefixo)
- Usar uma árvore completa (*full tree*)
todos os nós da árvore (excepto folhas) têm dois descendentes
- Minimiza o custo da codificação $\sum f_i d_i$
onde f_i é a frequência relativa e d_i é a profundidade na árvore

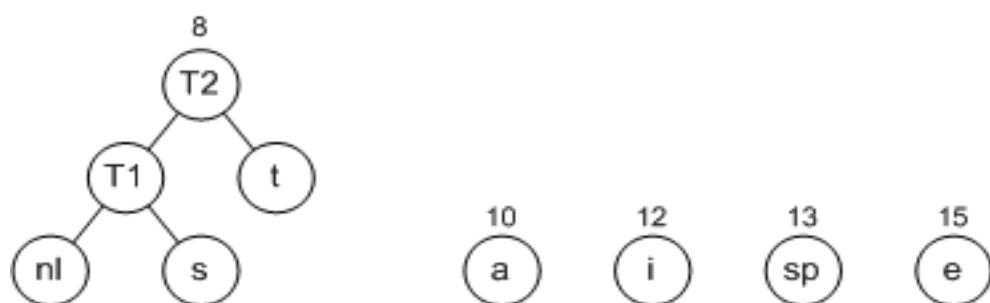
Algoritmo de Huffman

- O algoritmo de Huffman consiste de três passos básicos:
 - Cálculo da frequência de cada carácter no texto
 - Execução do algoritmo para construção de uma árvore binária
 - Codificação propriamente dita
- Inicialmente existe uma floresta de árvores só com raiz
- O peso de cada árvore é a soma das frequências relativas dos símbolos nas folhas
- Escolher as duas árvores com pesos menores e torná-las sub-árvores de uma nova raiz
- Repetir o passo anterior até haver uma só árvore
- Empates são resolvidos aleatoriamente

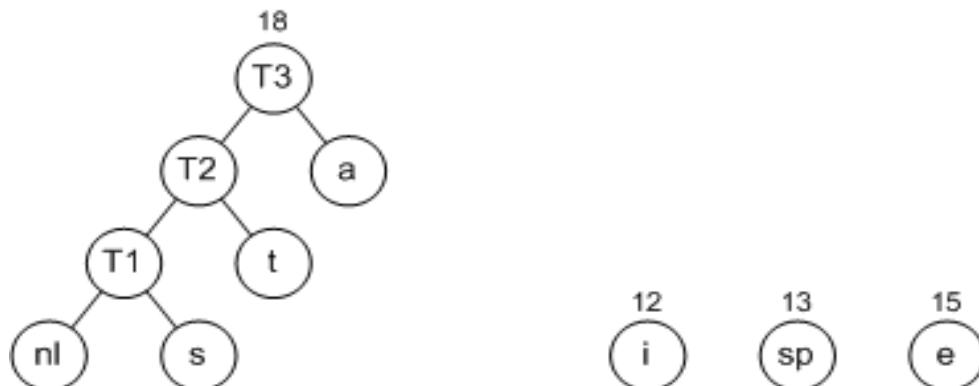
Algoritmo de Huffman (exemplo)



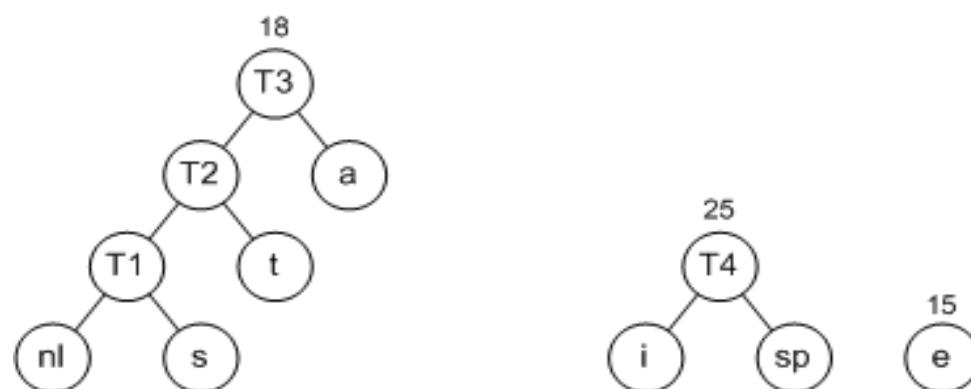
Algoritmo de Huffman (exemplo)



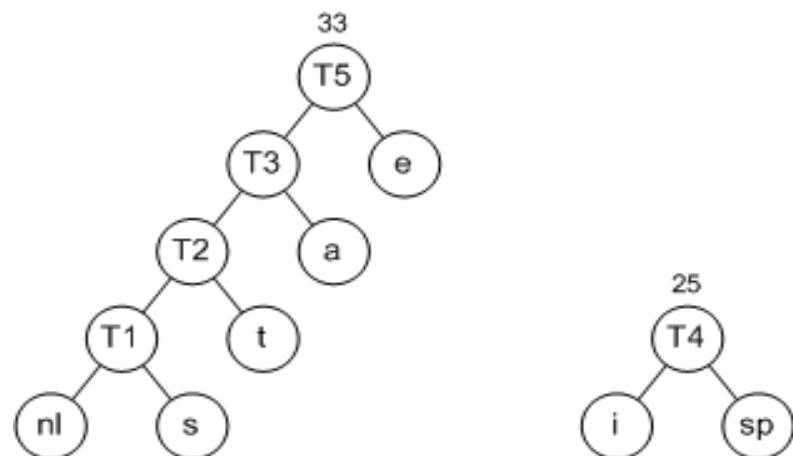
Algoritmo de Huffman (exemplo)



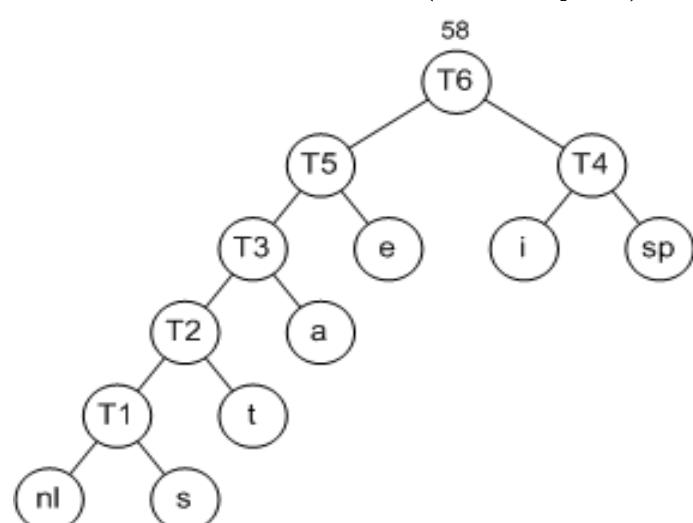
Algoritmo de Huffman (exemplo)



Algoritmo de Huffman (exemplo)



Algoritmo de Huffman (exemplo)



Algoritmo de Huffman

- Construção da árvore binária:

Algoritmo de Huffman:

Entrada: Um conjunto C de n caracteres.

Saída: Árvore de Huffman.

```

 $n \leftarrow |C|;$ 
 $Q \leftarrow C;$ 
para  $i \leftarrow 1$  até  $n - 1$  faça
    CriaNo(z);
     $x \leftarrow z.esq \leftarrow ExtraiMinimo(Q);$ 
     $y \leftarrow z.dir \leftarrow ExtraiMinimo(Q);$ 
     $f[z] \leftarrow f[x] + f[y];$ 
    Insere(Q, z);
retorne ExtraiMinimo(Q);

```

Referências e mais informação

- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
- David A. Huffman, *A Method for the Construction of Minimum-Redundancy Codes*, Proceedings of the Institute of Radio Engineers, 40(9):1098-1101
- Com base em slides de R. Camacho

Técnicas II

Técnicas de Concepção de Algoritmos: Branch and Bound

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
FEUP, MIEIC, CAL, 2010/2011

Branch and Bound

- O que é?
 - BB ou B&B é uma técnica de concepção de algoritmos genérica para encontrar soluções óptimas em vários problemas de optimização
 - Aplicada especialmente em problemas de optimização discreta e combinatorial
 - Aplica-se geralmente quando uma técnica gananciosa (*greedy*) ou programação dinâmica não são de todo apropriadas
 - É geralmente mais lento que as anteriores; pode levar a complexidades temporais exponenciais nos piores casos
 - Se bem aplicada, pode gerar soluções razoavelmente rápidas nos casos médios

Branch and Bound

■ Princípio

- Consiste na enumeração de todas as soluções candidatas
- Um alargado número de candidatos não “promissores” é descartado, estimando-se limites superiores e inferiores da quantidade a ser optimizada
- Baseia-se numa pesquisa em largura (*breadth-first search*), mas nem todos os vértices filhos são expandidos
 - Há um critério bem definido para dizer que filhos expandir e quando
 - Outro critério determina quando o algoritmo encontrou a solução óptima
- Método proposto por A. H. Land and A. G. Doig (1960) para programação discreta

A. H. Land and A. G. Doig (1960). "An automatic method of solving discrete programming problems". *Econometrica* **28** (3): pp. 497-520.

Aplicações

■ A abordagem é utilizada para resolver alguns problemas conhecidos, ditos “NP-duros” (*NP-hard*), como:

- Problema da mochila
- Programação Inteira
- Programação não-linear
- Problema do Caixeiro Viajante (*Travelling Salesman*) (TSP)
- Problema de afectação quadrática (QAP)
- *The Maximum Satisfiability Problem* (MAX-SAT)
- Pesquisa do Vizinho mais Próximo (NNS)
- O problema de corte de stock
- False Noise Analysis

B&B: Algoritmo Geral

- Considerações

- Cada solução é assumida poder ser expressa como um array X[1:n], como por exemplo em técnicas de retrocesso
- Um “preditor”, ou função de custo aproximado CC, deve ser definida

- Definições importantes

- Um “live node” é um vértice que não foi expandido
- Um “dead node” é um vértice que já foi expandido
- Um “expanded node” (ou “e-node”) é um “live node” com melhor CC

B&B: Algoritmo Geral

```

Procedure B&B()
begin
    E: nodepointer;
    E := new(node); -- this is the root node which
                      -- is the dummy start node
    H: heap;          -- A heap for all the live nodes
                      -- H is a min-heap for minimization problems,
                      -- and a max-heap for maximization problems.
    while (true) do
        if (E is a final leaf) then
            -- E is an optimal solution
            print out the path from E to the root;
            return;
            endif

        Expand(E);
        if (H is empty) then
            report that there is no solution;
            return;
        endif
        E := delete-top(H);
    endwhile
end

```

B&B: Algoritmo Geral

```

Procedure Expand(E)
begin
    - Generate all the children of E;
    - Compute the approximate cost value CC of each child;
    - Insert each child into the heap H;
end

```

Funções de Custo Aproximado

- Como escolher?
 - **Definição** da Função de Custo C : para cada vértice x na árvore de solução, a função de custo $C(x)$ é o custo da melhor solução que vai até o vértice x .
 - **Teorema:**
No caso de problemas de minimização, se $CC(x) \leq C(x)$ para todo vértice x , e se $CC(x) = C(x)$ para todo vértice folha x , então o primeiro vértice a ser expandido (melhor CC), que é um vértice folha, corresponde à solução óptima do problema!
 - *Prove!*

Funções de Custo Aproximado

- Critérios para escolha da função de custo aproximado CC
 - Para problemas de minimização:
 $CC(x) \leq C(x)$ para todo “live node” x
 $CC(x) = C(x)$ para todo vértice folha
 - Para problemas de maximização:
 $CC(x) \geq C(x)$ para todo “live node” x
 $CC(x) = C(x)$ para todo vértice folha
 - CC é chamada uma subestimação de C (no caso de minimização), e uma superestimação de C (no caso de maximização)
 - Quanto mais próximo que CC estiver de C , mais rápido será capaz o algoritmo de encontrar a solução óptima!

Referências e mais informação

- “Introduction to Algorithms”, Second Edition, Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, The MIT Press, 2001
- “The Algorithm Design Manual”, Steven S. Skiena, Springer-Verlag, 1998
- A. H. Land and A. G. Doig (1960). “An automatic method of solving discrete programming problems”. *Econometrica* 28 (3): pp. 497-520

Técnicas de Concepção de Algoritmos: Branch and Bound: Exemplo

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
 FEUP, MIEIC, CAL, 2010/2011

Exemplo de Aplicação: B&B

■ Problema

- Uma companhia está a montar uma equipa para levar a cabo uma série de operações
- Há **quatro membros** na equipa: A, B, C e D
- Há **quatro operações** a realizar: 1, 2, 3, e 4
- Cada membro da equipa pode realizar exactamente uma operação, e Todas as quatro operações devem ser executadas com sucesso, para que para que o projecto todo seja considerado concluído satisfatoriamente.
- Entretanto, a probabilidade de um membro particular da equipa ser bem sucedido na realização de uma operação particular varia, como indicado na tabela abaixo:

Team member	A	Operation			
		1	2	3	4
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

Exemplo de Aplicação: B&B

■ Problema (cont...)

- Se aos membros da equipa fossem atribuídas as tarefas ABCD, nesta ordem, então a probabilidade do projecto ser concluído na sua totalidade com sucesso seria:

$$(0.9)(0.6)(0.85)(0.7) = 0.3213 \sim 32\%$$
- **Pressuposto:** Se houver formas possíveis de organizar a equipa de modo a se obter uma taxa de sucesso para o projecto como um todo que exceda aos **45%**, então o gestor irá aprovar o projecto!
- **Questão:** para esta data equipa, o gestor irá aprovar o projecto?

Team member	Operation			
	1	2	3	4
A	0.9	0.8	0.9	0.85
B	0.7	0.6	0.8	0.7
C	0.85	0.7	0.85	0.8
D	0.75	0.7	0.75	0.7

Exemplo de Aplicação: B&B

■ Formulação:

- Vértices na árvore: uma atribuição pessoa-tarefa, total ou parcial
- Política de selecção de vértices: melhor valor global da **função de custo aproximado** (*bounding function*)
- Política de selecção de variáveis: escolher a próxima operação na ordem natural, ou seja 1 a 4
- Função de Custo (**bounding function**): para operações não atribuídas, escolher o melhor membro da equipa que esteja disponível, ou seja, o que possui maior probabilidade de sucesso, ainda que a pessoa seja escolhida mais do que uma vez
- Função Objectivo (critério de paragem): quando um estado-solução candidato tiver o valor resultante da função e custo maior do que o valor do estado-candidato corrente
- Estado-solução candidato: quando a função de custo retorna uma solução em que cada operação é atribuída a uma pessoa diferente

Team member	Operation			
	1	2	3	4
A	0.9	0.8	0.9	0.85
B	0.7	0.6	0.8	0.7
C	0.95	0.7	0.85	0.8
D	0.75	0.7	0.75	0.7

Solução candidata: ?

No início, nenhuma decisão foi tomada;
Calcula-se o valor máximo do potencial
de sucesso para o projecto:

Se A for escolhido para realizar as 4
operações, então:

$$\text{AAAA} = (0.9)(0.8)(0.9)(0.85) = 0.5508$$

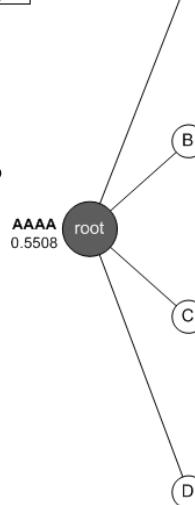
0.5508 root

Team member	Operation			
	1	2	3	4
A	0.9	0.8	0.9	0.85
B	0.7	0.6	0.8	0.7
C	0.85	0.7	0.85	0.8
D	0.75	0.7	0.75	0.7

Solução candidata: ?

Para a Operação 1, expande-se o
vértice raiz (branching process).

Cada nó filho corresponde à atribuição
da primeira atribuição a um membro
específico da equipa...



		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.95	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

Solução candidata: ?

Neste nó, A, decidiu-se que a pessoa A irá realizar a operação 1. Iremos indicar isso colorindo A em verde

A não pode ser selecionada para realizar qualquer outra tarefa, porque já foi escolhido

```

graph TD
    root((root  
AAAA  
0.5508)) --> A((A))
    root --> B((B))
    root --> C((C))
    root --> D((D))
  
```

FEUP Universidade do Porto Faculdade de Engenharia

Branch and Bound - CAL, 2010/11

7

		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

Solução candidata: ?

As outras pessoas serão atribuídas aos outros trabalhos. Isso será feito com o mesmo critério utilizado para os outros: a pessoa com melhor probabilidade será escolhida, ainda que se repita; se houver impasse, escolhe-se arbitrariamente

O resultado da função de custo para este estado ADCC, dá-nos um valor de:

$ADCC = (0.9)(0.7)(0.85)(0.8) = 0.4284$

```

graph TD
    root((root  
AAAA  
0.5508)) --> A((ADCC  
0.4284))
    root --> B((B))
    root --> C((C))
    root --> D((D))
  
```

FEUP Universidade do Porto Faculdade de Engenharia

Branch and Bound - CAL, 2010/11

8

		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.95	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

Solução candidata: ?

Passa-se para o próximo nó, em que a decisão é que B realize a operação 1; neste caso, B não poderá realizar qualquer outra operação; as restantes pessoas serão atribuídas às outras operações;

Neste caso, A é a melhor pessoa para realizar 2, 3, e 4, resultando em:

$$\text{BAAA} = (0.7)(0.8)(0.9)(0.85) = 0.4284$$

Realiza-se o mesmo procedimento para os outros nós filhos a serem explorados.

FEUP Universidade do Porto Faculdade de Engenharia

Branch and Bound - CAL, 2010/11

9

		Operation			
		1	2	3	4
Team member	A	0.9	0.8	0.9	0.85
	B	0.7	0.6	0.8	0.7
	C	0.85	0.7	0.85	0.8
	D	0.75	0.7	0.75	0.7

Solução candidata: ?

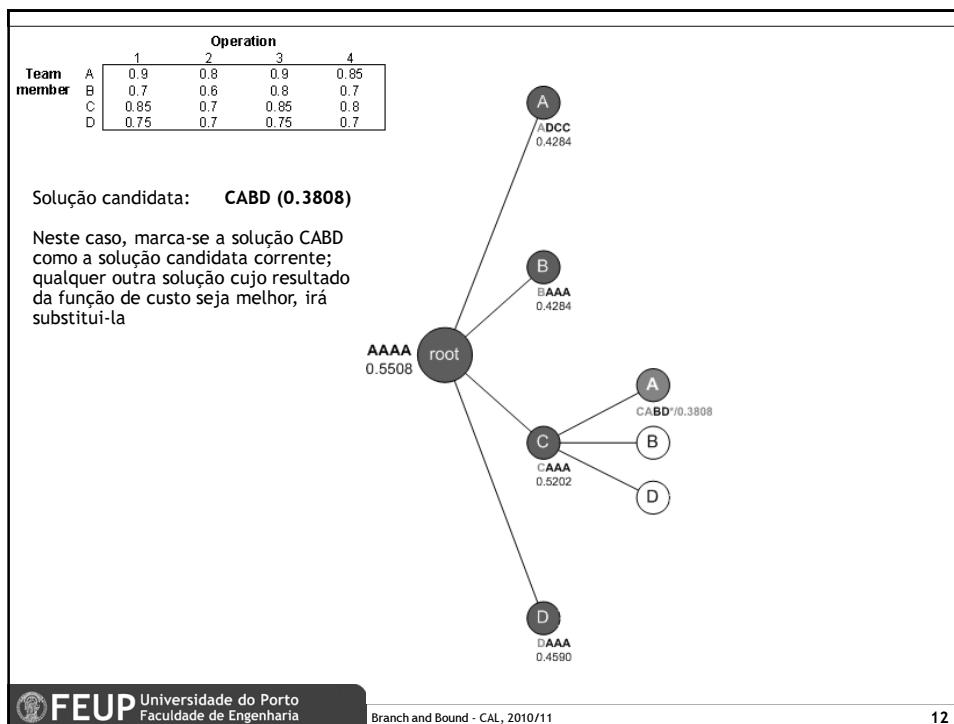
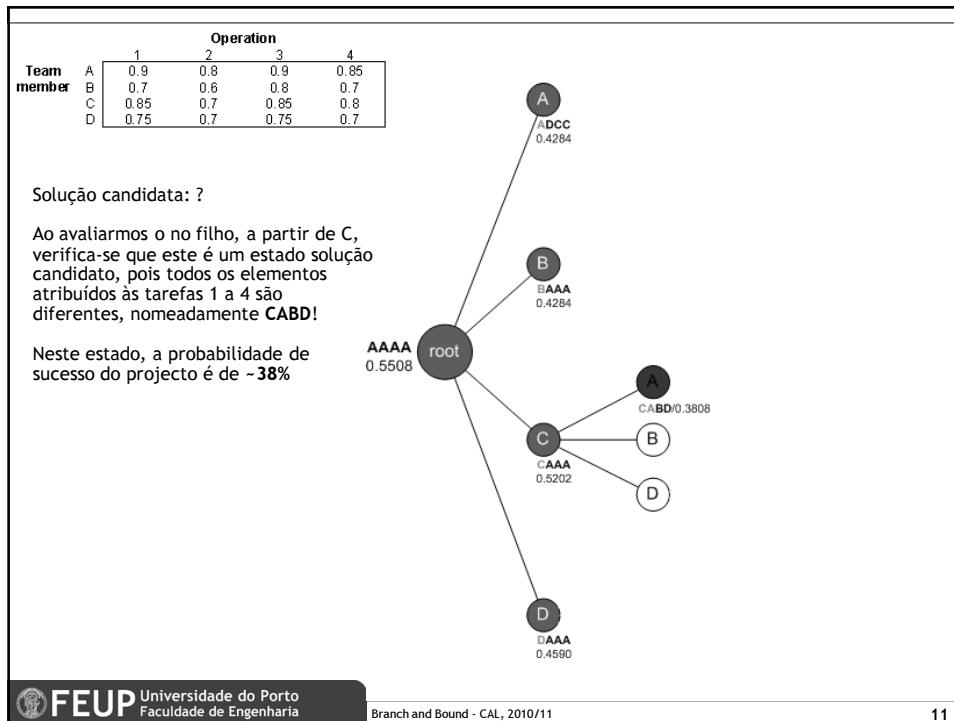
Agora iremos expandir (branch) os nós da árvore.

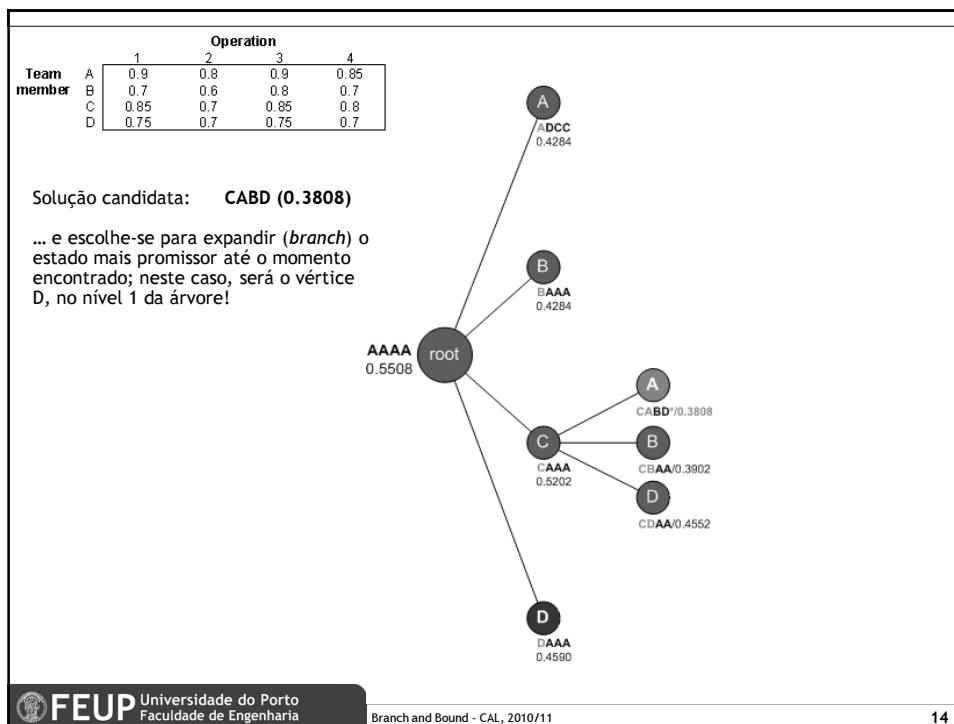
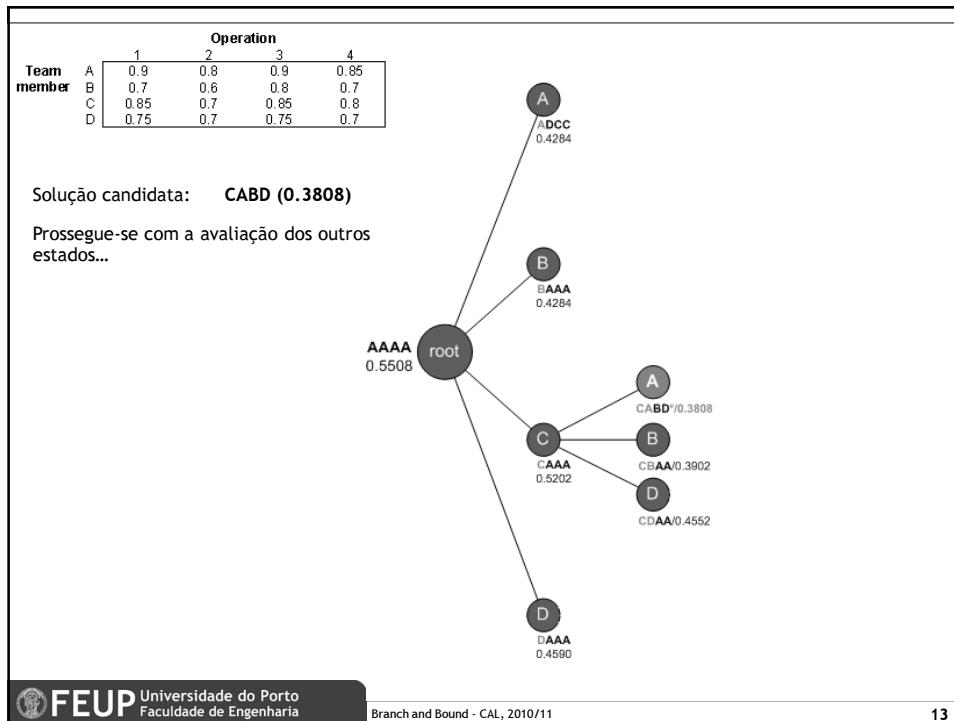
O nó que escolhemos expandir é o nó que é mais promissor, ou seja, aquele que potencialmente irá levar-nos à melhor solução. Neste caso é o nó C, com valor **0.5202**

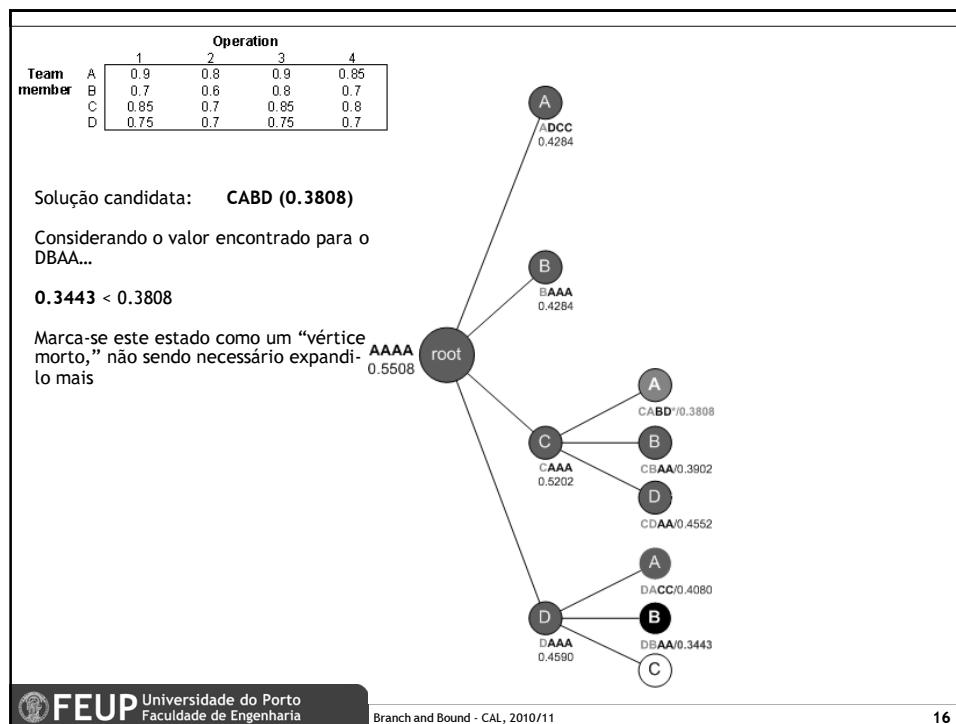
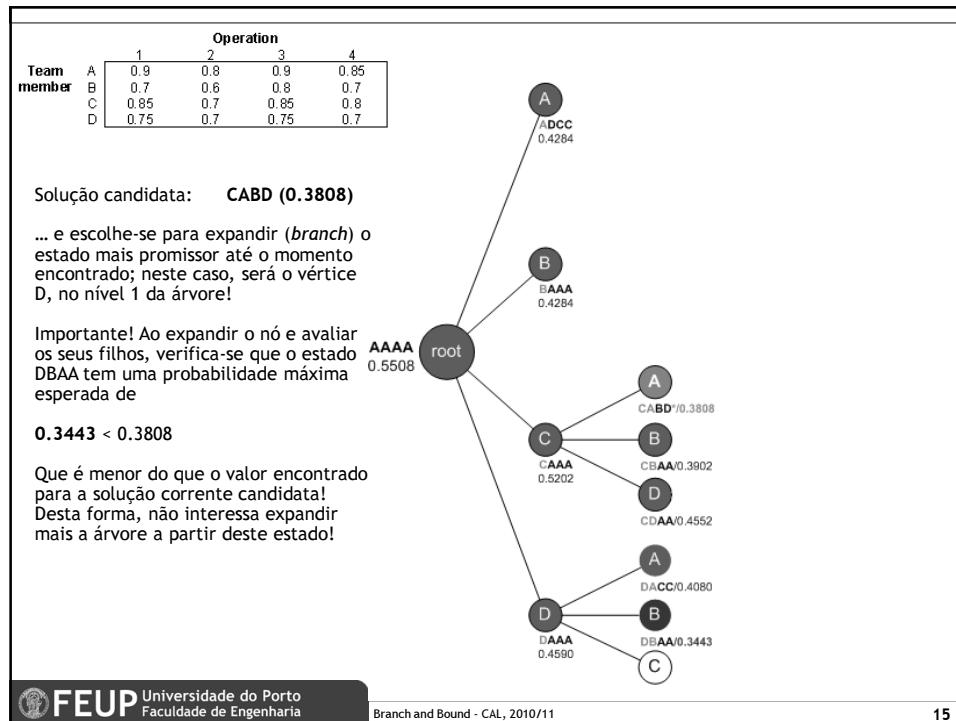
FEUP Universidade do Porto Faculdade de Engenharia

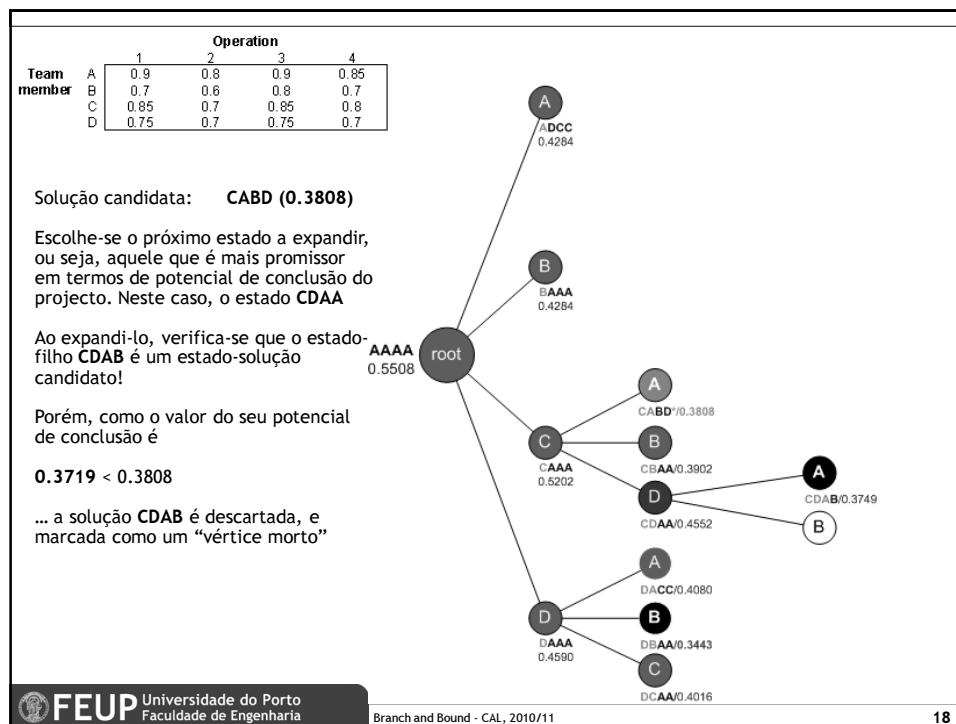
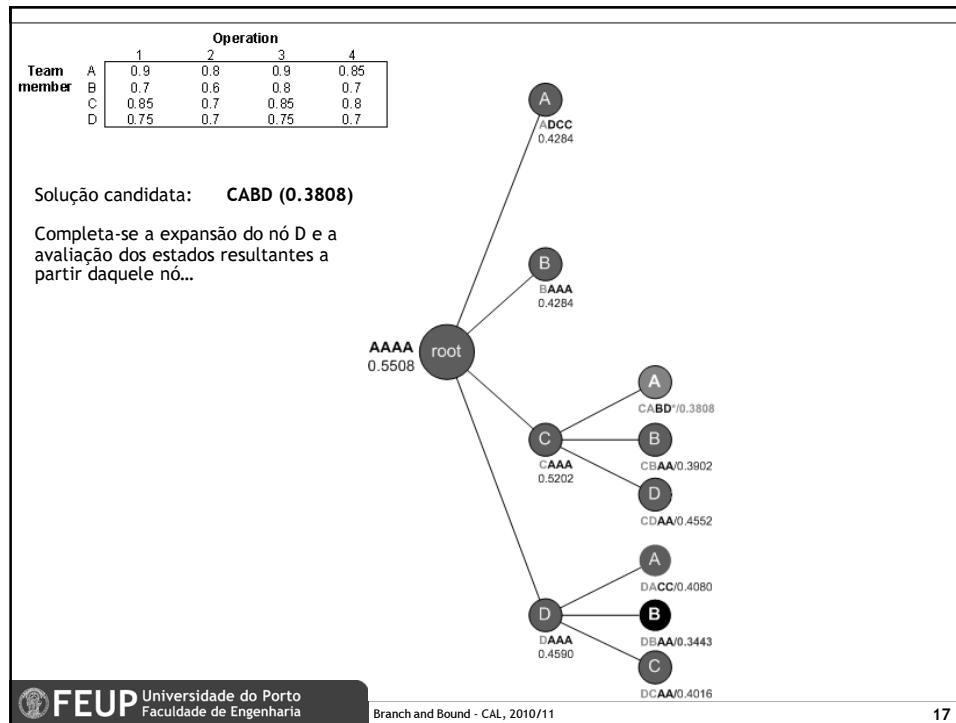
Branch and Bound - CAL, 2010/11

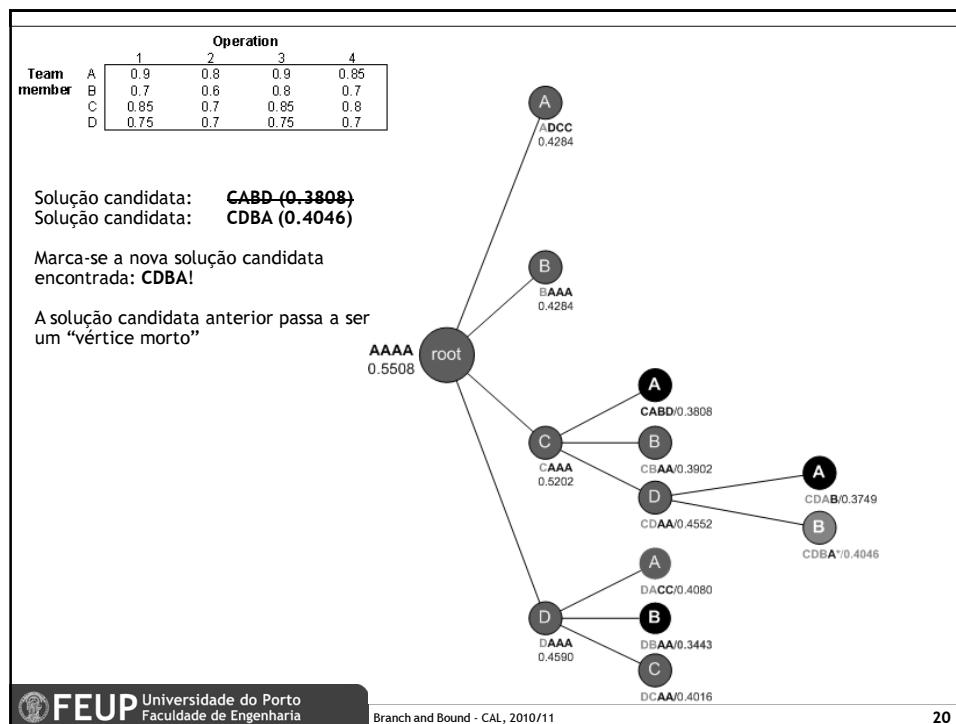
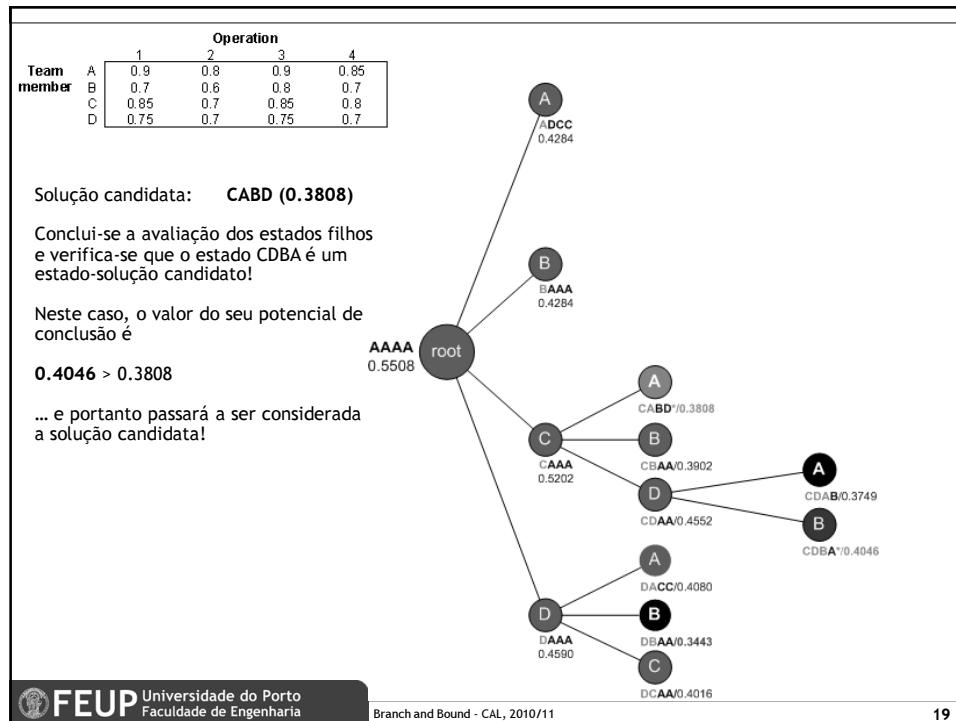
10

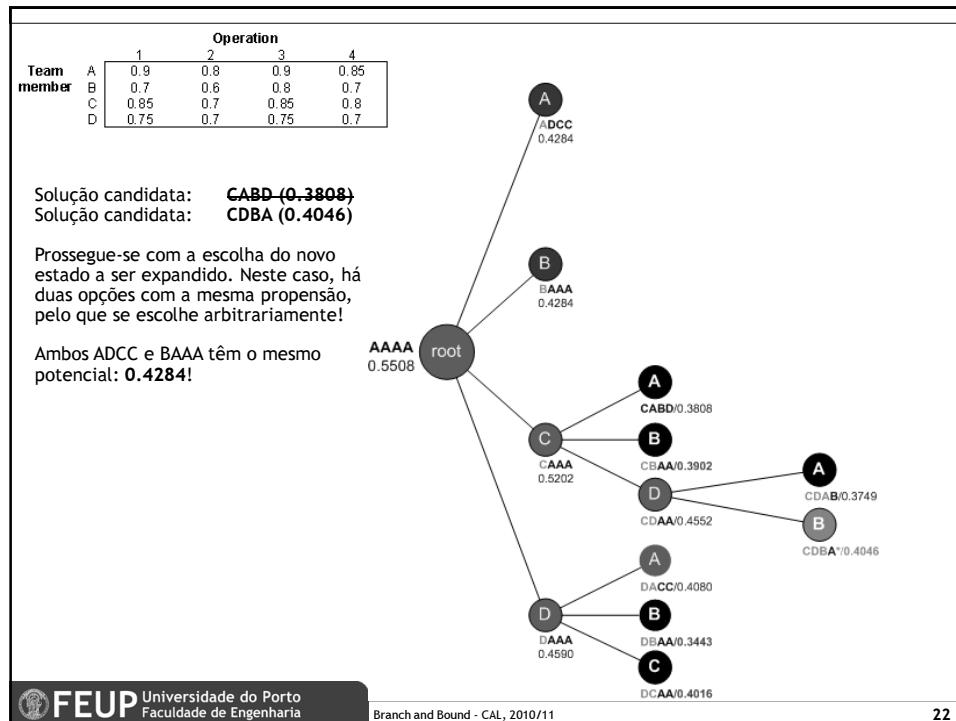
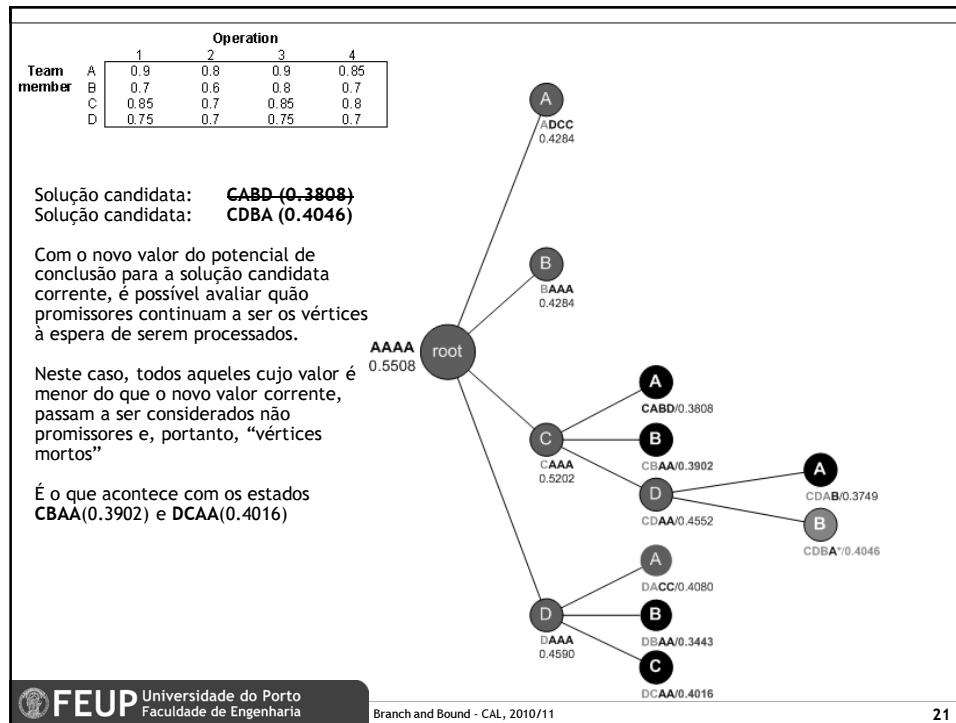


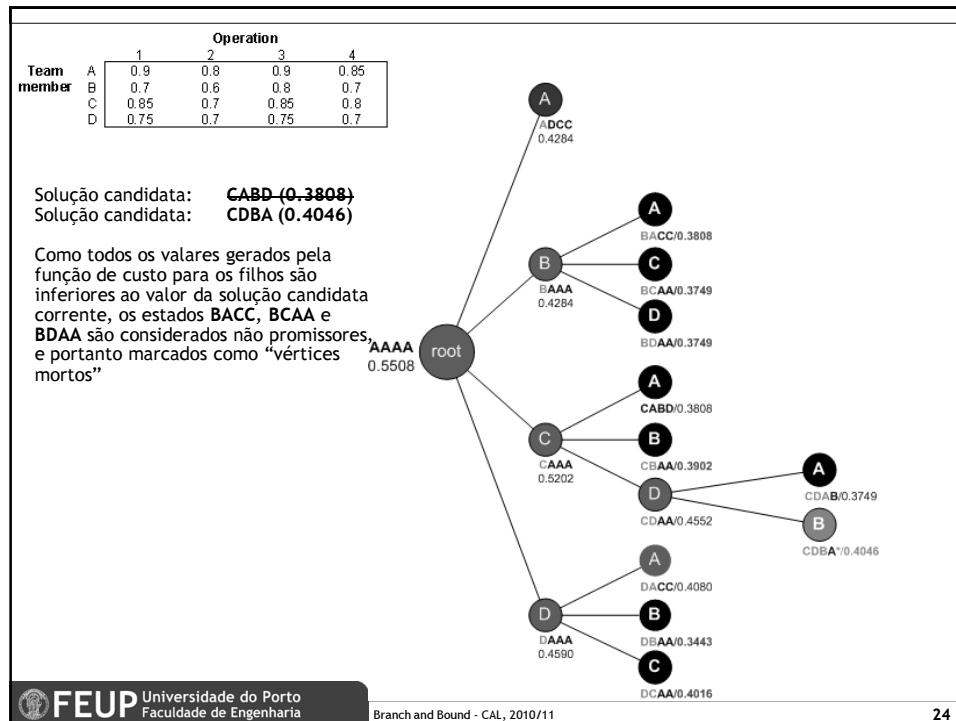
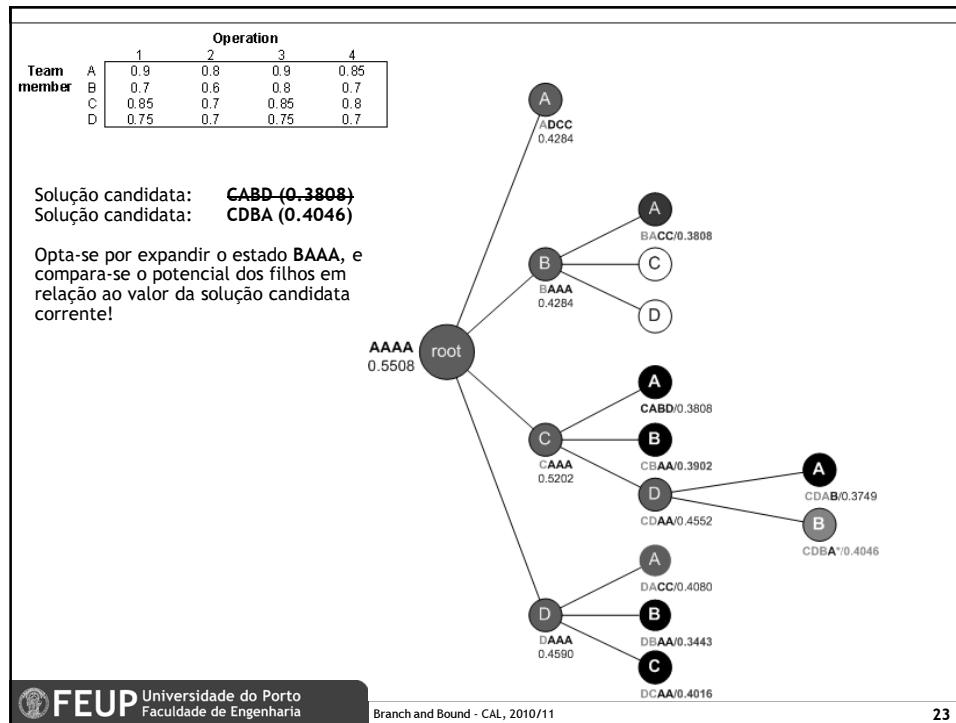


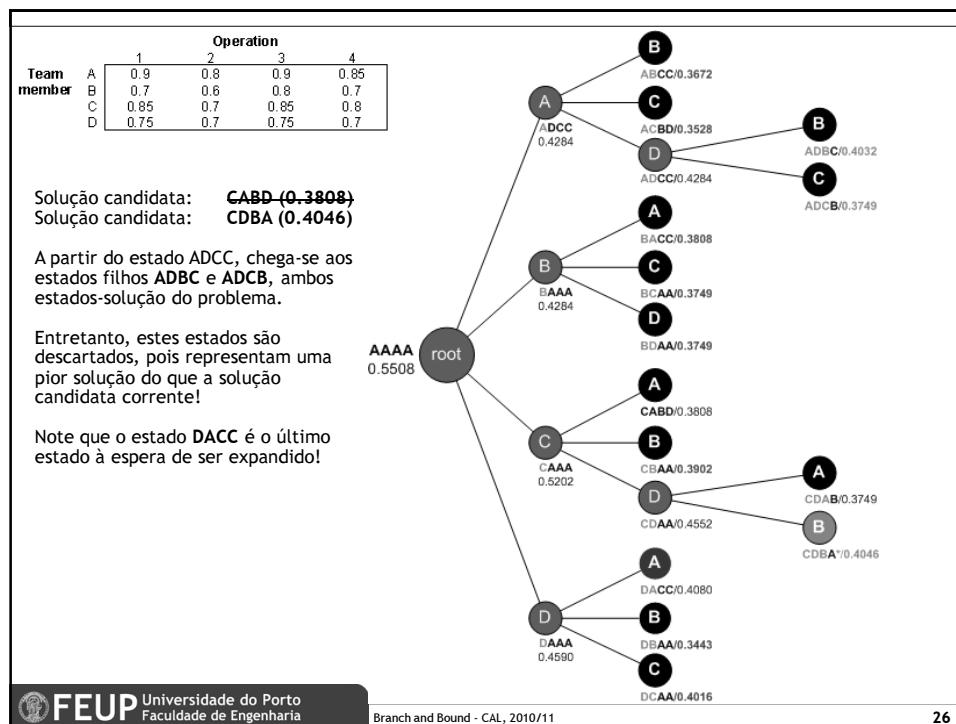
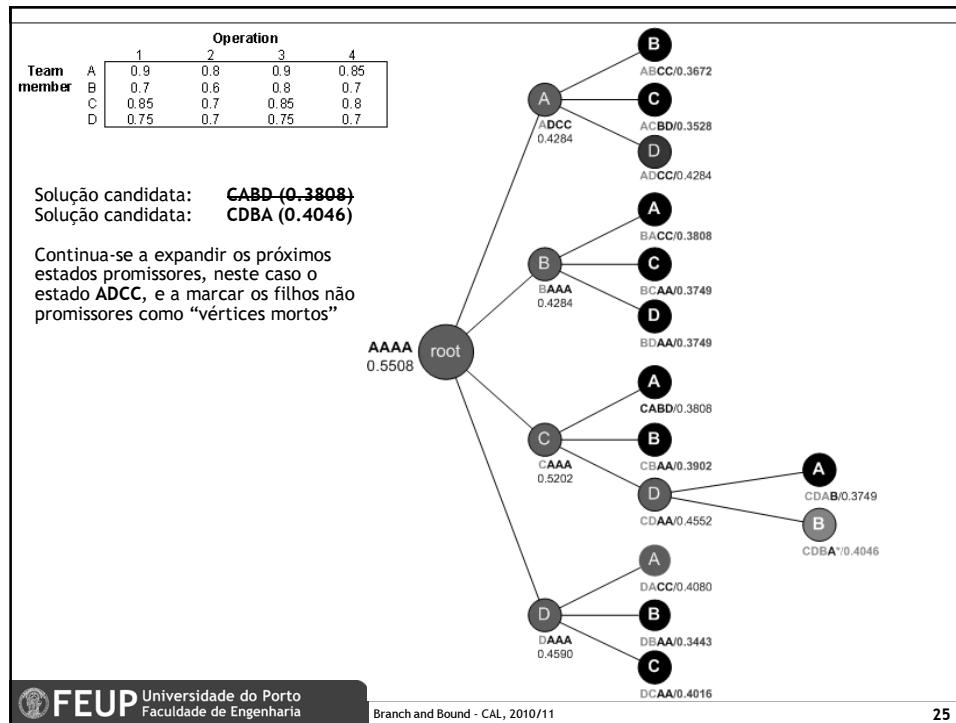


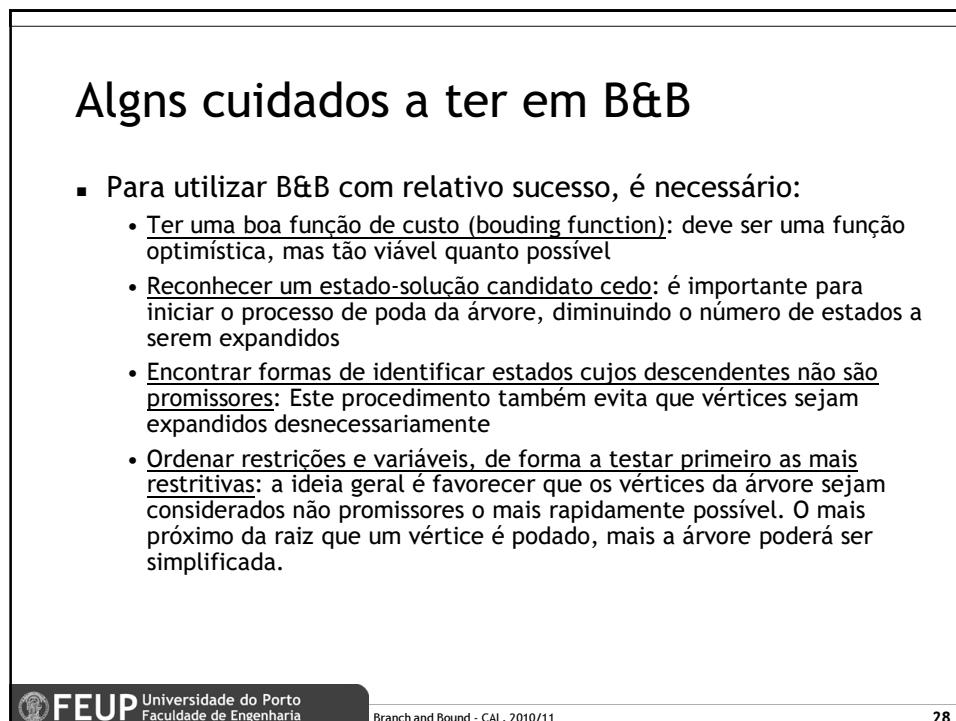
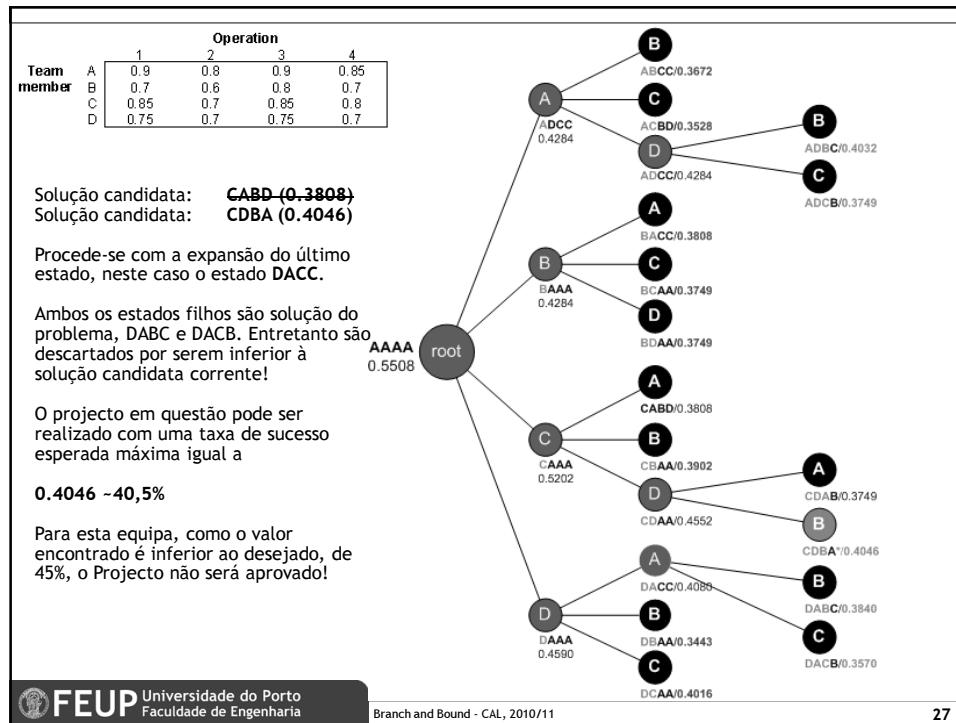












Referências e mais informação

- John W. Chinneck (2010) “Practical Optimization: a Gentle Introduction”, Carleton University, CA.

Técnicas de Concepção de Algoritmos: Algoritmos Aleatórios

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes
FEUP, MIEIC, CAL, 2010/2011

Algoritmos Determinísticos

■ Características

- π - um problema a resolver (por computador); e.g. Ordenação (*sorting*)
- A - um algoritmo para resolver o problema π ; e.g. Selection Sort
- A cada ponto da execução do algoritmo A sobre um input I , a próxima acção de A é bem definida e inequívoca
- A sua execução e o tempo que leva a correr, os passos intermediários e o resultado final são os mesmos durante cada execução de A sobre I
- O curso seguido pelo algoritmo não varia com a sua execução, desde que o input I permaneça inalterado

■ Problemas

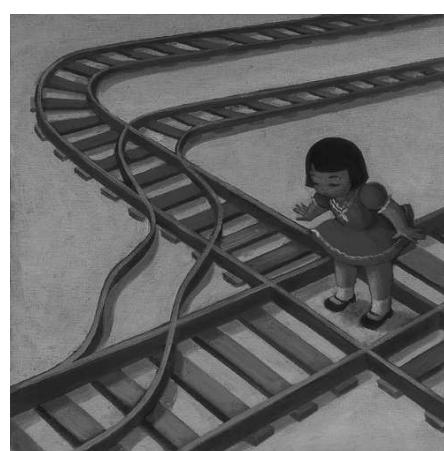
- Pode ser difícil formular um algoritmo com bom tempo de execução
- Aumento da complexidade (temporal e/ou espacial) com o $|input|$

Quando tudo mais falha...

- ... relaxa-se algumas premissas:
 - Respostas devem ser exactas?
 - Apenas uma instância do problema está a ser considerada?
 - A plataforma utilizada é sequencial?
 - A plataforma utilizada é determinística?
- Métodos alternativos:
 - Algoritmos de aproximação
 - Algoritmos Off-line
 - Algoritmos paralelos e distribuídos
 - **Algoritmos aleatórios**
 - Algoritmos que podem estar errados, mas com baixa probabilidade
 - Heurísticas

O que são algoritmos aleatórios?

- Analogia com o processo de tomada de decisão...

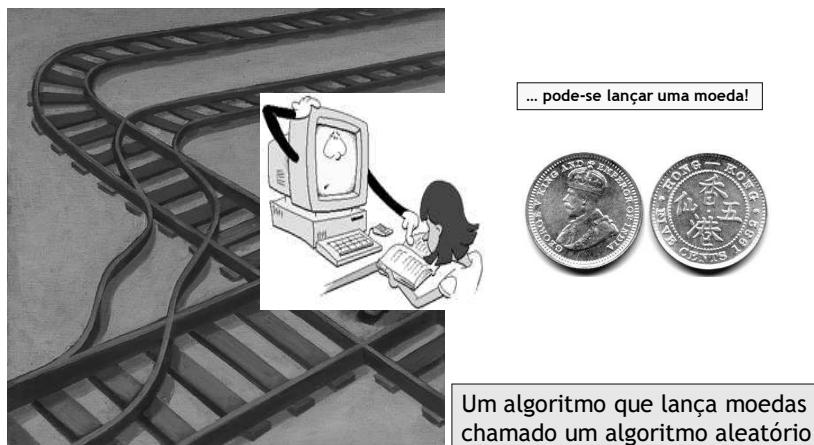


... pode-se lançar uma moeda!



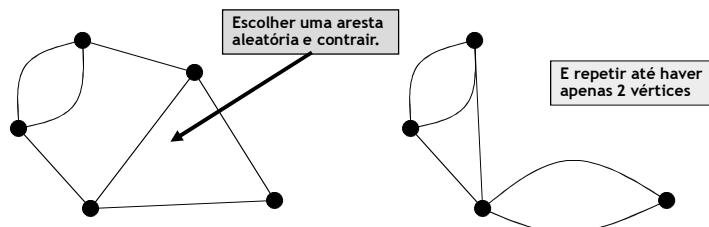
O que são algoritmos aleatórios?

- Analogia com o processo de tomada de decisão...



Por que aleatoriedade?

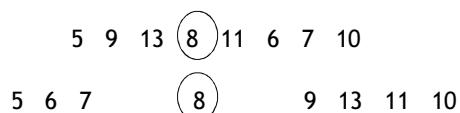
- Tomar decisões pode ser complicado:**
-> Um algoritmo aleatório pode ser mais simples!
- Problema do corte mínimo em grafos:**
 - Pode ser resolvido pelo método do fluxo máximo
 - Introduzindo aleatoriedade...



Por que aleatoriedade?

- Tomar decisões pode ser muito caro:
-> Um algoritmo aleatório pode ser mais rápido!

- Problema de ordenação de n elementos:



Escolher um elemento no centro torna o procedimento muito eficiente,
mas é caro encontrar tal elemento (i.e. tempo linear).

Escolher um elemento aleatoriamente poderá resolver perfeitamente!

Por que aleatoriedade?

- Premissa do Método Probabilístico:
 - > Provar a existência de um objecto que satisfaça certas propriedades sem necessariamente precisar construir tal objecto!
 - > Demonstrar que um objecto aleatório satisfaz tais propriedades com uma certa probabilidade!
- Em alguns problemas é necessário tratar uma vasta quantidade de dados, e muitas vezes não se pode aceder a tal informação em tempo útil!
- Algoritmos sublineares -> aleatoriedade é essencial!

Mais formalmente...

■ Definição

- Um algoritmo aleatório é um algoritmo que aplica algum grau de aleatoriedade como parte da sua lógica de funcionamento
- Isto induz à introdução de algum grau de incerteza no funcionamento do algoritmo também
- Esta classe de algoritmos tipicamente utiliza *bits* de input aleatórios (uniformemente distribuídos) como principal elemento de condução do seu comportamento/funcionamento
- O objectivo é se conseguir atingir um desempenho bom para o “caso médio” sobre todas as possíveis escolhas no conjunto de opções aleatórias
- Formalmente, o desempenho do algoritmo é uma variável aleatória determinada como função dos *bits* de input aleatórios considerados; portanto, tempo de execução ou resultado (ou ambos) são variáveis aleatórias

Mais formalmente...

■ Como gerar aleatoriedade?

- Na prática, algoritmos aleatórios são aproximados e implementados a partir da utilização de geradores de **números ditos pseudo-aleatórios** no lugar de fontes de *bits* verdadeiramente aleatórios
- Geradores de número pseudo-aleatórios devem ser utilizados com discrição; algumas considerações devem ser tidas em conta, a fim de se evitar situações em que o resultado gerado desvie consideravelmente do resultado teórico esperado

Diferenças importantes

■ Algoritmos Aleatórios:

- Utilizam input aleatório com o objectivo de melhorar o desempenho em termos de tempo (de execução) e espaço (de memória);
- Sempre produz um resultado correcto para o período de tempo estipulado

■ Algoritmo Probabilísticos:

- a depender do input aleatório, podem potencialmente produzir um resultado incorrecto ou mesmo não produzir qualquer resultado - ou por assinalar uma falha, ou por não conseguir terminar de todo
- No caso dos algoritmos probabilísticos, o termo “algoritmo” é questionável, especialmente para os casos em que um resultado aceitável ou satisfatório nunca é produzido
- Entretanto, em alguns problemas, tais algoritmos são a única forma de se tentar chegar a uma solução

Diferenças importantes

■ Algoritmos “Las Vegas”:

- Algoritmo aleatório que garantidamente retorna uma resposta correcta
- O seu tempo de execução é uma variável aleatória cuja expectativa é delimitada (e.g. polinomialmente)

■ Algoritmos “Monte Carlo”:

- Algoritmo aleatório que pode retornar uma resposta incorrecta, ou mesmo não retornar qualquer resultado para o problema
- O algoritmo é executado por um número predefinido de passos, k , e é esperado produzir uma resposta que é correcta com probabilidade $\geq 1/3$

- Em ambos os casos as probabilidades/expectativas são consideradas apenas sobre as escolhas aleatórias feitas pelo algoritmo - i.e. independentes do *input*

Exemplo

■ Problema

- Encontrar um elemento ‘a’ num *array* de n elementos
- **Input:** um *array* de n elementos em que metade dos elementos são *a*’s e a outra metade são *b*’s
- **Output:** encontrar a posição relativa no *array* de um elemento ‘*a*’

■ Solução “Las Vegas”

```
findingA_LV(array A, n)
begin
repeat
    Randomly select one element out of n elements.
    until 'a' is found
end
```

- Probabilidade de sucesso do algoritmo é de 1 (100%)
- Tempo de execução é aleatório e a sua expectativa é limitada a $O(1)$

Exemplo

■ Solução “Monte Carlo”

```
findingA_MC(array A, n)
begin
    i=1
repeat
    Randomly select one element out of n elements.
    i = i + 1
until i=k
end
```

- Se um elemento ‘*a*’ for encontrado, o algoritmo é bem sucedido, caso contrário, o algoritmo falha
- Depois de k execuções, a probabilidade de se encontrar um ‘*a*’ é:
 $\Pr[\text{find_a}] = 1 - 1(1 / 2)^k$
- Este algoritmo não garante sucesso, mas o tempo de execução é fixo. A iteração é executada exactamente k vezes e, portanto, a complexidade temporal é $O(k)$

Paradigmas dos Algoritmos Aleatórios

- Frustrando o adversário (*Foiling the adversary*)
 - Aplica-se a problemas que podem ser vistos como um jogo entre quem concebe o algoritmo e um adversário. O adversário tem uma função de recompensa, que o algoritmo tenta minimizar
- Abundância de testemunhas (*Abundance of witnesses*)
 - Alguns problemas são do tipo “o input i tem a propriedade p ?”. Tipicamente, a propriedade de interesse pode ser estabelecida pela definição de um objecto, chamado “testemunha”. Em alguns casos, encontrar “testemunhas” deterministicamente é difícil!
 - Se a selecção de amostragens repetidas produzir uma “testemunha” então tem-se uma prova matemática de que o i tem propriedade p
- Verificação de identidade (*verifying an identity*)
 - $f(x_1, \dots, x_n) \equiv 0$? Pode-se gerar um vector aleatório (a_1, \dots, a_n) e verificar se $f(a_1, \dots, a_n) \equiv 0$

Paradigmas dos Algoritmos Aleatórios

- Ordenação aleatória do input (*random ordering of input*)
 - Em algoritmos convencionais, o desempenho poderá depender da ordem prévia dos dados de input; utilizando-se um algoritmo aleatório, essa dependência é removida. Quicksort, no pior caso, corre em tempo $O(n^2)$ e quando utiliza um algoritmo aleatório realiza em $O(n \log n)$
- *Fingerprinting*
 - Esta técnica tenta representar um objecto mais complexo por uma *fingerprint*. Em circunstâncias apropriadas, de dois objectos terem a mesma *fingerprint*, então há uma forte evidência de serem idênticos. Apropriado para analisar correspondência de padrões (*pattern matching*)
- Quebra de simetria (*Symmetry breaking*)
 - Útil em processamento distribuído, quando vários processadores devem colectivamente decidir sobre uma acção (aparentemente indistinta) entre várias outras acções

Paradigmas dos Algoritmos Aleatórios

■ *Rapidly mixing Markov Chains*

- Útil para problemas de contagem, tais como contar o número de ciclos de um grafo, ou o número de árvores, ou correspondências (*matchings*), ou outros problemas similares
- Primeiro, o problema de contagem é transformado num problema de amostragem. Cadeias de Markov podem ser utilizadas para gerar pontos de um dado espaço aleatoriamente, mas que precisam convergir rapidamente
- Uma cadeia de Markov é um processo estocástico (aleatório) que evolui no tempo, formado basicamente por:

Um conjunto de estados (assumido ser finito): $1, \dots, N$

Uma matriz de transições P onde p_{ij} representa a probabilidade de se mover para o estado j quando se está no estado i

Algumas aplicações

- Algoritmos sobre teorias numéricas (e.g. número primos)
- Manipulação de estruturas de dados
- Identidades algébricas (e.g. verificação matriz identidade)
- Programação matemática (e.g. programação linear)
- Algoritmos em grafos (e.g. cortes mínimos e cortes máximos)
- Contagens e enumerações
- Computação distribuída e paralela
- Existência probabilística de provas

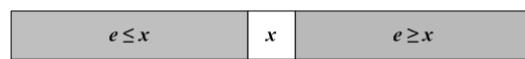
Exemplo: Quicksort

- Factos

- Proposto por C. A. R. Hoare em 1962
- É um algoritmo baseado em “Divisão e Conquista”
- Realiza ordenação “in-place”
(como “insertion sort”, mas diferente de “merge sort”)
- Muito prático (se for bem ajustado)

- Princípio (p/ ordenar array de n elementos):

- Dividir: partir o array em 2 subarrays em torno de um pivot x , tal que



- Conquistar: recursivamente ordenar os dois subarrays
- Combinar: os dois subarrays são concatenados

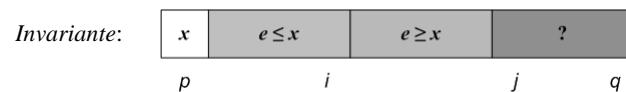
Exemplo: Quicksort

- Subrotina de partição:

```

Partition( $A, p, q$ )  $\diamond A[p..q]$ 
   $x \leftarrow A[p]$             $\diamond$  pivot =  $A[p]$ 
   $i \leftarrow p$ 
  for  $j \leftarrow p + 1$  to  $q$ 
    do if  $A[j] \leq x$ 
      then  $i \leftarrow i + 1$ 
      exchange  $A[i] \leftrightarrow A[j]$ 
    exchange  $A[p] \leftrightarrow A[i]$ 
  return  $i$ 

```



Exemplo: Quicksort

- Pseudocódigo para o Quicksort

```
Quicksort(A, p, r)
  if p < r
    then q ← Partition(A, p, r)
        Quicksort(A, p, q - 1)
        Quicksort(A, q + 1, r)
```

Chamada inicial: Quicksort(A, 1, n)

- Análise, no pior caso:

- $T(n)$: complexidade no pior caso para o Quicksort aplicado sobre um input de n elementos. Apenas comparações são contabilizadas
- $T(n) = \max\{ T(\pi) : \pi \text{ é uma permutação de } [n] \}$
- $T(n) = \Theta(n^2)$
- No pior caso o input é : $\pi = \langle n, n - 1, \dots, 1 \rangle$

Exemplo: Quicksort Aleatório

- Ideia: Partição em torno de um elemento aleatório

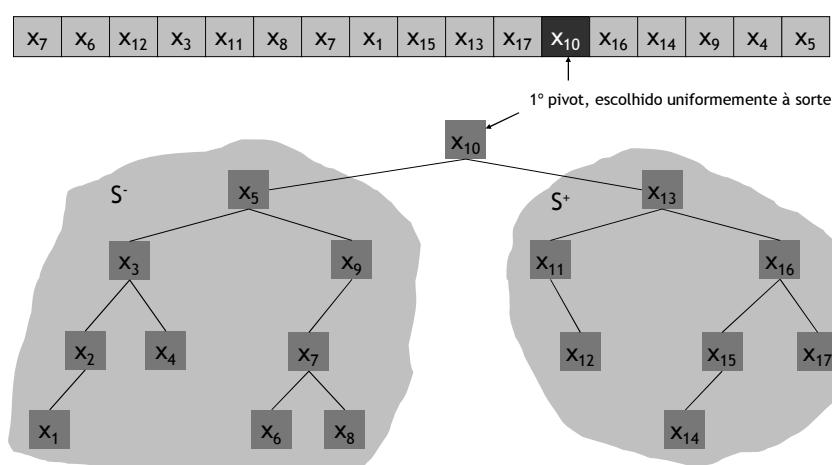
- Tempo de execução é independente da ordem do *input*
- Não é necessário fazer-se qualquer pressuposto a respeito da distribuição dos elementos do *input*
- Nenhum *input* específico implica no seu comportamento no pior caso
- O pior caso é determinado apenas como resultado da saída de um gerador de números aleatórios
- **O pivot é o elemento aleatório do algoritmo!**
- Esta solução “protege” o algoritmo de ser executado no pior caso!

Exemplo: Quicksort Aleatório

- RandQSort (A, p, q)
 - If $p \geq q$, EXIT
 - Seleccionar uniformemente ao acaso $r \in \{p, \dots, q\}$
 - $s \leftarrow$ posição correcta de $A[r]$ no vector ordenado (# elem < $A[r]$)
 - Mover aleatoriamente o pivot escolhido $A[r]$ para a posição s
 - Mover os elementos restantes para as posições “*apropriadas*”
 - RandQSort($A, p, s - 1$)
 - RandQSort($A, s + 1, q$)

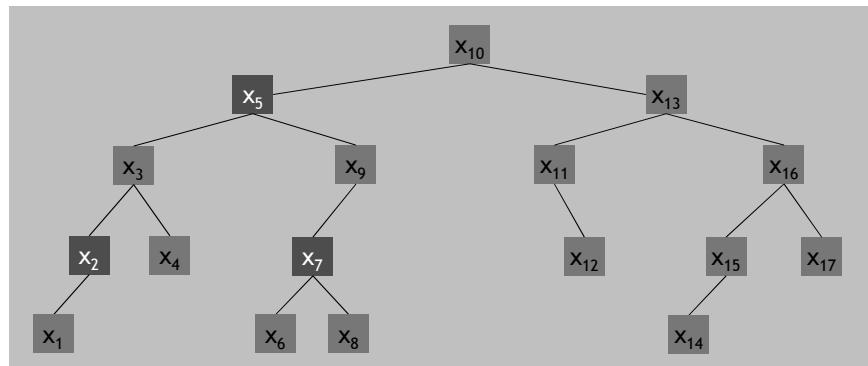
Exemplo: Quicksort Aleatório

- Representação BST da selecção dos pivots



Exemplo: Quicksort Aleatório

- Observação: elementos são apenas comparados com os seus ancestrais ou descendentes
- $\Pr[x_i \text{ and } x_j \text{ are compared}] = 2 / |j - i + 1|$



Exemplo: Quicksort Aleatório *

■ Análise:

- Toda comparação é realizada entre o pivot e um outro elemento
- Dois elementos são comparados no máximo uma vez
- O rank de um elemento é a sua posição na estrutura ordenada
- Seja x_i o elemento do rank i . $S_{i,j} = \{x_i, \dots, x_j\}$

$$X_{i,j} = \begin{cases} 1, & \text{se } x_i \text{ e } x_j \text{ são comparados} \\ 0, & \text{caso contrário} \end{cases}$$

Exemplo: Quicksort Aleatório *

- Análise:

- $E[T(\pi)] = E[\sum_{i < j} X_{i,j}] = \sum_{i < j} E[X_{i,j}]$
- $E[X_{i,j}] = 2 / (j - i + 1)$
- $E[T(\pi)] = \sum_{i < j} 2 / (j - i + 1)$

$$\sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1} = 2 \sum_{i=1}^n \sum_{j=2}^i \frac{1}{j} \leq 2n \sum_{j=1}^n \frac{1}{j} \approx 2n \int_{x=1}^n \frac{1}{x} dx = 2n \ln n$$

↑
probabilidade de i e j serem comparados

- O número esperado de comparações é $O(n \log n)$!
- * Ver discussão em Cormen *et al.* (2009) “Introduction to Algorithms”

“Derandomization”

- Primeiro constrói-se um algoritmo aleatório para resolver um problema mais eficientemente, depois tenta-se realizar um processo de “derandomization” a fim de produzir um algoritmo determinístico para resolver o mesmo problema.

Geração de Números Aleatórios

- Propriedades dos números aleatórios

- Cada número aleatório R_i é uma amostra independente tirada de uma distribuição uniforme contínua entre zero e 1.

$$f(x) = \begin{cases} 1, & 0 \leq x \leq 1 \\ 0, & \text{caso contrário} \end{cases}$$

- O valor expectável de cada R_i , é dado por:

$$E(R) = \int_0^1 x dx = \frac{x^2}{2} \Big|_0^1 = \frac{1}{2}$$

- E a sua variância é dada por:

$$V(R) = \int_0^1 x^2 dx - [E(R)]^2 = \frac{x^3}{3} \Big|_0^1 - \left(\frac{1}{2}\right)^2 = \frac{1}{3} - \frac{1}{4} = \frac{1}{12}$$

Geração de Números Aleatórios

- Consequências das propriedades de uniformidade e independência

- Se o intervalo (0, 1) for dividido em n classes, ou subintervalos de igual tamanho, o número esperado de observações em cada intervalo é N/n , onde N é o número total de observações
- A probabilidade de se observar um valor em um intervalo particular é independente dos valores anteriormente retirados da amostra

Geração de Números Aleatórios

- Problemas que podem ocorrer com a geração de número pseudo-aleatórios
 - Os números gerados podem não ser uniformemente distribuídos
 - Os números podem ser gerados num intervalo discreto e não num intervalo contínuo
 - A média dos números gerados pode ser muito alta ou muito baixa
 - A variância dos números gerados pode ser muito alta ou muito baixa
 - Pode haver variações cíclicas, como por exemplo
 - (a) autocorrelação entre números
 - (b) números sucessivamente muito mais altos ou muito mais baixos do que os números adjacentes
 - (c) muitos números acima da média, seguidos de muitos números abaixo da média

Geração de Números Aleatórios

- Algumas considerações na implementação de rotinas para geração de números pseudo-aleatórios
 - O procedimento deve ser rápido
 - O procedimento deve ser portável para diferentes arquitecturas e diferentes linguagens de programação
 - O algoritmo deve gerar um ciclo de números suficientemente longo
 - Os números aleatórios gerados devem poder ser replicados
 - Os números aleatórios gerados devem obedecer, tanto quanto possível, as propriedades estatísticas ideais de uniformidade e independência

Geração de Números Aleatórios

- Técnicas: *Método da Congruência Linear*

- Proposto por Lehmer, em 1951
- Produz uma sequência de inteiros, X_1, X_2, \dots , entre zero e $m - 1$, seguindo a relação recursiva

$$X_{i+1} = (aX_i + c) \bmod m, \quad i = 0, 1, 2, \dots$$

- O valor inicial X_0 é chamado *seed* (semente), a é uma constante multiplicadora, c é o incremento e m é o módulo.
- Variações da técnica:
Mixed Congruential Method: se $c \neq 0$
Multiplicative Congruential Method: se $c = 0$
- A selecção dos valores de a , c , m , e X_0 afectam drasticamente as propriedades estatísticas e o tamanho do ciclo resultante

Geração de Números Aleatórios

- Técnicas: *Método dos geradores combinados*

- Um período da ordem de $2^{31} - 1 \approx 2 \times 10^9$ não é mais adequado para a maioria das aplicações
- Para máquinas de 32-bit, L'Ecuyer, em 1988, sugeriu combinar dois geradores:

$$(1) \quad m_1 = 2147483563, \quad a_1 = 40014$$

$$(2) \quad m_2 = 2147483399, \quad a_2 = 40692$$

Geração de Números Aleatórios

- Técnicas: *Método dos geradores combinados*

1. Seleccionar a “semente” $X_{1,0}$ no intervalo $[1, 2147483562]$ para o primeiro gerador, e a “semente” $X_{2,0}$ no intervalo $[1, 2147483398]$

2. Calcular cada gerador individual

$$X_{1,j+1} = 40014 X_{1,j} \text{ mod } 2147483563$$

$$X_{2,j+1} = 40692 X_{2,j} \text{ mod } 2147483399$$

3. Combinar $X_{j+1} = (X_{1,j+1} + X_{2,j+1}) \text{ mod } 2147483562$

4. Retornar

$$R_{j+1} = \begin{cases} \frac{X_{j+1}}{2147483563}, & X_{j+1} > 0 \\ \frac{2147483562}{2147483563}, & X_{j+1} = 0 \end{cases}$$

5. Atribuir $j = j + 1$ e repetir a partir de 2...

o período do gerador combinado é $(m_1 - 1)(m_2 - 1)/2 \approx 2 \times 10^{18}$

Referências e mais informação

- Richard M. Karpa (1991) “An Introduction to Randomized Algorithms”, *Discrete Applied Mathematics*, 34:165-201.
- Juraj Hromkovic (1998) “Design and Analysis of Randomized Algorithms: Introduction to Design Paradigms”, Berlin: Springer.
- T. Cormen *et al.* (2009) “Introduction to Algorithms”, Cambridge, MA: MIT press.
- R. Sedgewick (1978) Implementing quicksort programs, *Comm. ACM*, 21(10):847-857.
- J. Banks *et al.* (1996) *Discrete-event System Simulation*. Upper Saddle River, NJ: Prentice-Hall

Problemas intratáveis

Introdução à classe de problemas NP- Completos

R. Rossetti, A.P. Rocha, A. Pereira, P.B. Silva, T. Fernandes

FEUP, MIEIC, CAL, 2010/2011

Introdução

■ Considerações Práticas

- Em alguns casos práticos, alguns algoritmos podem resolver problemas simples em tempo razoável (e.g. $n \leq 20$); mas quando se trata de *inputs* maiores (e.g. $n \geq 100$) o desempenho degrada consideravelmente
- Soluções desse género podem estar a executar em tempo exponencial, da ordem de $n^{\log n}$, 2^n , 2^{2^n} , $n!$, ou mesmo piores do que isso
- Para algumas classes de problemas, é difícil determinar se há algum paradigma ou técnica que leve à solução do mesmo, ou se há formas de provar que o problema é intrinsecamente difícil, não sendo possível encontrar uma solução algorítmica cujo desempenho seja sub-exponencial
- Para alguns problemas difíceis, é possível afirmar que, se um desses problemas se pode resolver em tempo polinomial, então todos podem ser resolvidos em tempo polinomial!

Tempo Polinomial como referência

- Tempo polinomial é a referência que define e separa a classe de problemas que **podem ser resolvidos eficientemente**. Assim, se um problema pode ser resolvido eficientemente, então significa que o seu tempo de execução é polinomial.
- Esta avaliação é geralmente medida em termos do tempo de execução do algoritmo, usando a complexidade no pior caso, como uma função de n , que é o tamanho do *input* do problema
- Um algoritmo de tempo polinomial tem tempo de execução da ordem de $O(n^k)$, onde k é uma constante independente de n
- Um problema é dito ser “**resolúvel em tempo polinomial**” se houver um algoritmo de tempo polinomial que o resolva
- Algumas funções parecem não ser polinomiais, mas podem ser tratadas como tal: e.g. $O(n \log n)$ tem delimitação superior da ordem de $O(n^2)$
- Algumas funções parecem ser polinomiais, mas podem não o ser na verdade: e.g. $O(n^k)$, se k variar em função de n , tamanho do *input*.

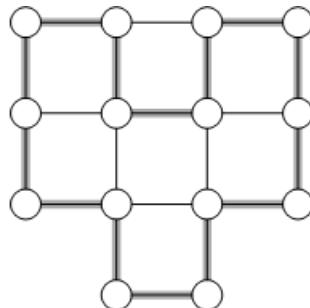
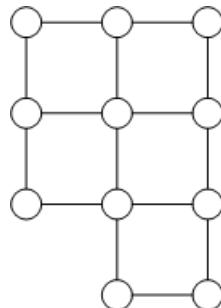
Problemas de Decisão

- Reformulação de problemas de optimização
 - Muitos dos problemas práticos que se pretende resolver são problemas de optimização (maximizar ou minimizar alguma métrica)
 - Um problema é dito ser um “**problema de decisão**” se o seu *output* ou resposta deve ser um simples “SIM” ou “NÃO” (ou derivativos do tipo “V/F”, “0/1”, “aceitar/rejeitar”, etc.)
 - Muitos problemas de optimização podem ser expressos em termos de problemas de decisão.

Por exemplo: o problema “qual o menor número de cores que se pode utilizar para colorir um grafo?” pode ser expresso como “Dado um grafo G e um inteiro k , é possível colorir G com k cores?”
- A **classe de problemas P** é definida por todos os problemas de decisão que podem ser resolvidos em tempo polinomial!

Verificação do Tempo Polinomial

- *Undirected Hamiltonian cycle problem (UHC)*
 - Dado um grafo G , “é possível determinar se G possui um ciclo que visita todo vértice exactamente uma vez?”



Verificação do Tempo Polinomial

- Caso se conheça um ciclo (e.g. $\langle v_3, v_7, v_1, \dots, v_{13} \rangle$), é fácil verificá-lo, por inspecção. Ainda não sendo possível implementar algoritmo p/resolver o problema, seria fácil “verificar” se G é ou não “Hamiltoniano”
- O ciclo neste caso é dito ser um “certificado”; trata-se de uma informação que permite verificar se uma dada “string” está numa “linguagem” (em problemas de reconhecimento de linguagem)
- Caso seja possível verificar a precisão de um certificado para um problema em tempo polinomial, diz-se que o problema é “verificável em tempo polinomial”
- Nem todas as “línguas” gozam da propriedade de serem facilmente verificáveis! Por exemplo: seja o problema definir se um grafo G tem exactamente um ciclo de Hamilton. É fácil verificar se existe pelo menos um ciclo, mas não é simples demonstrar que não há outro!
- A classe de problemas NP é definida por todos os problemas que podem ser verificados por um algoritmo de T polinomial

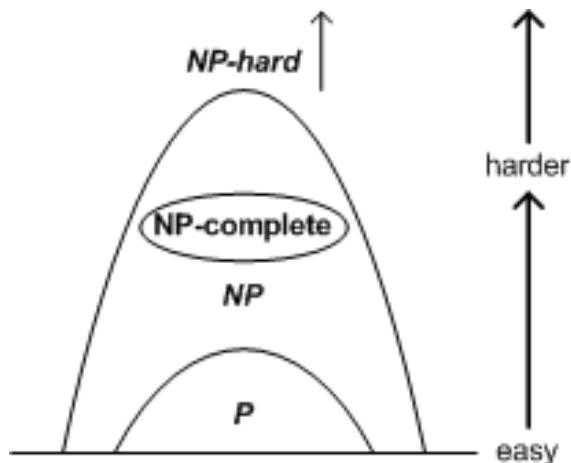
Verificação do Tempo Polinomial

- Execução de tempo polinomial é diferente de verificação de tempo polinomial!
 - O circuito de Hamilton é verificável em tempo polinomial, mas acredita-se não haver uma solução executável em tempo polinomial que encontre um circuito de Hamilton
- Por que NP e não VP?
 - O termo NP vem de “*nondeterministic polynomial time*,” relacionado com um programa a executar num “computador não determinístico” capaz de realizar palpites; basicamente, tal arquitectura seria capaz de não deterministicamente conjecturar o valor do certificado e verificar, em tempo polinomial, se uma *string* está na linguagem ou não.

Classes P e NP

- $P \subseteq NP$.
 - Assim, se um problema é resolúvel em tempo polinomial, então pode-se certamente verificar se uma solução é correcta em tempo polinomial
- Não se sabe certamente se $P = NP$.
 - Ou seja, poder verificar se uma solução é correcta em tempo polinomial não garante ou ajuda encontrar um algoritmo que resolva o problema em tempo polinomial
 - $P \neq NP$? Muitos autores acreditam que sim, mas não há provas!
- A classe de problemas **NP-Completos** é a classe dos problemas mais difíceis de resolver em toda a classe NP.
- Pode haver problemas ainda mais difíceis de resolver que não estejam enquadrados na classe de problemas NP; neste caso, são chamados **problemas NP-difíceis (NP-hard)**

Classes P e NP



Redução de Problemas NP

- Suponha que há dois problemas, A e B . Sabe-se que A é impossível de ser resolvido em tempo polinomial
- Pretende-se provar que B não pode ser resolvido em tempo polinomial. Como provar ou demonstrar que:

$$(A \notin \mathbf{P}) \Rightarrow (B \notin \mathbf{P})$$
- Pode-se tentar provar o contraposto:

$$(B \in \mathbf{P}) \Rightarrow (A \in \mathbf{P})$$
 - Em outras palavras, para demonstrar que B não é resolúvel em tempo polinomial, supõem-se que há um algoritmo que resolve B em tempo polinomial, e então deriva-se uma contradição pela demonstração de que A pode ser resolvido em tempo polinomial

Redução de Problemas NP

- Premissa

- Suponha que há uma subrotina que pode resolver qualquer instância do problema B em tempo polinomial
- Tenta-se demonstrar que a mesma subrotina pode ser utilizada para resolver A em tempo polinomial
- Então tem-se “reduzido o problema A no problema B ”!
- Como se sabe que A não se pode resolver em tempo polinomial, então está-se basicamente a tentar provar que a subrotina não pode existir, implicando que B não pode ser resolvido em tempo polinomial

Redução de Problemas NP

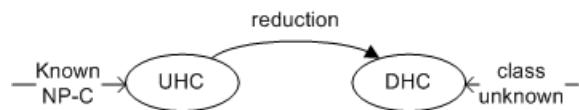
- Exemplo

- Sabe-se que o problema UHC é um problema NP-completo! Não se conhece algoritmo de tempo polinomial que o resolva.
- Problema: suponha um director técnico pede a um dos seus engenheiros para encontrar uma solução polinomial para um problema diferente, nomeadamente o problema de encontrar um ciclo de Hamilton num grafo dirigido (DHC).
- Depois de pensar numa solução, por algum tempo, o engenheiro convence-se de que não se trata de uma solicitação sensata. Será que considerar arestas direcccionais tornaria o problema de alguma forma mais fácil? Apesar de ambos (director e eng.) concordarem que UHC é NP-completo, o director está convencido de que DHC é viável e fácil. Como convencê-lo do contrário?

Redução de Problemas NP

- Solução

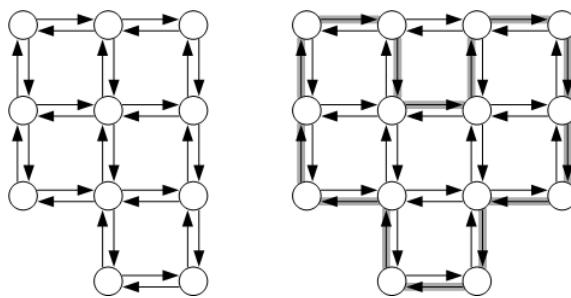
- Pode-se tentar convencer o director de que “se houvesse uma solução eficiente para DHC, então se demonstraria que seria então possível resolver UHC em tempo polinomial.”
- Em particular, usar-se-ia a mesma subrotina usada no DHC para resolver o UHC. Uma vez que ambos conhecem que UHC não é resolúvel, então tal rotina não pode existir. Portanto, DHC também não é resolúvel em tempo polinomial!



Redução de Problemas NP

- Solução

- Dado um grafo G , criar um grafo dirigido G' pela substituição de cada aresta $\{u, v\}$ por duas arestas dirigidas: (u, v) e (v, u)
- Cada caminho simples em G é um caminho simples em G' , e vice-versa. Portanto, G terá um ciclo de Hamilton se, e somente se, G' também o tiver!



Redução de Problemas NP

- Solução

- Redução UHC → DHC

```
bool UHC (graph G) {
    create digraph G' with the same number of vertices as G
    foreach edge (u, v) in G
        Add edges (u, v) and (v, u) in G'
    return DHC (graph G')
}
```

- Note-se que nenhum dos problemas foi efectivamente resolvido. Apenas demonstrou-se como converter uma solução para o DHC numa solução para o UHC. Este procedimento é chamado “**redução**” e é crucial para a teoria dos problemas NP-completos.

Redução de Problemas NP

- Definição

- Dados dois problemas, A e B , diz-se que A é polinomialmente redutível a B se, dada uma subrotina de tempo polinomial para B , pode-se utilizá-la para resolver A em tempo polinomial. Quando tal se verifica, expressa-se por

$$A \leq_p B$$

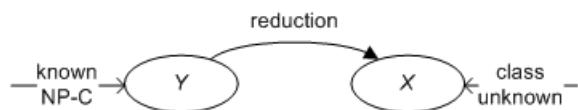
- **Lema:** Se $A \leq_p B$ e $B \in P$ então $A \in P$
- **Lema:** Se $A \leq_p B$ e $A \notin P$ então $B \notin P$
- **Lema:** Se $A \leq_p B$ e $B \leq_p C$ então $A \leq_p C$ (transitividade)

Redução de Problemas NP

- Definição +formal da classe dos problemas NP-completos
 - Um problema de decisão $B \in \text{NP}$ é NP-completo se $A \leq_p B \mid \forall A \in \text{NP}$
 - Assim, se B pode ser resolvido em tempo polinomial, então qualquer outro problema A em NP é resolúvel em tempo polinomial
 - **Lema:** B é NP-completo se
 - (1) $B \in \text{NP}$, e
 - (2) $A \leq_p B$ para algum problema A , se A é NP-Completo

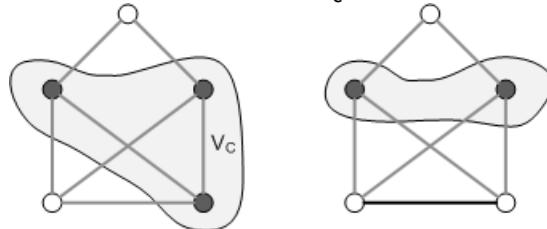
Redução de Problemas NP

- Procedimento:
Dado um problema X , prove que o mesmo é pertencente à classe dos problemas NP-completos.
 - Provar que X está em NP
 - Selecionar um problema Y que se sabe ser NP-completo
 - Definir uma redução de tempo polinomial de Y para X
 - Provar que, dada uma instância de Y , Y tem uma solução se, e somente, X tem uma solução



Vertex Cover

- Uma cobertura de vértices de um grafo $G = (V, E)$ é um subconjunto $V_C \subseteq V$, tal que toda aresta $(a, b) \in E$ é incidente em pelo menos um vértice $u \in V_C$.

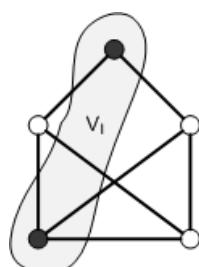


- Vértices em V_C “cobrem” todas as arestas em G .
- Reformulação de VC como um problema de decisão
 - O grafo G tem uma cobertura de vértices de tamanho k ?

Independent Set

- Um conjunto independente de um grafo $G = (V, E)$ é um subconjunto $V_I \subseteq V$, tal que não há dois vértices em V_I que partilham uma aresta de E

- $u, v \in V_I$ não podem ser vizinhos em G .



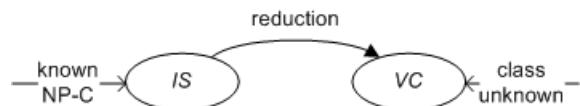
- Reformulação de VI como um problema de decisão
 - O grafo G tem um conjunto independente de tamanho k ?

Vertex Cover é NP-completo?

- Dado que o problema de decisão de Conjunto Independente (IS) é NP-completo, provar que o problema de Cobertura de Vértices também é NP-completo
- Solução: (1) Provar que VC pertence à classe NP.
 - Dado V_C , uma cobertura de vértices de $G = (V, E)$, $|V_C| = k$
Pode-se verificar em $O(|E| + |V|)$ que V_C é uma cobertura de vértices de G . Como?
 - Para cada vértice $\in V_C$, remover todas as arestas incidentes
 - Verificar se todas as arestas foram removidas de G .
 - Então, *Vertex Cover* $\in NP$!

Vertex Cover é NP-completo?

- (2) Selecionar um problema que se conhece ser NP-completo
 - O problema do Conjunto Independente (IS), em grafos, é reconhecidamente um problema *NP*-completo!
 - Usar IS para provar que VC é *NP*-completo



Vertex Cover é NP-completo?

- (3) Definir uma redução de tempo polinomial de IS para VC:
 - Dada uma instância geral de IS: $G' = (V', E')$, k'
 - Construir uma instância específica de VC: $G = (V, E)$, k
 - $V = V'$
 - $E = E'$
 - $(G = G')$
 - $k = |V'| - k'$
 - Esta transformação é polinomial:
 - Tempo constante para construir $G = (V, E)$
 - $O(|V|)$ para contar o número de vértices
 - Provar que há um V_I ($|V_I| = k'$) para G' se, e se somente, há um V_C ($|V_C| = k$) para G .

Vertex Cover é NP-completo?

- (3) Definir uma redução de tempo polinomial de IS para VC:
 - Dada uma instância geral de IS: $G' = (V', E')$, k'
 - Construir uma instância específica de VC: $G = (V, E)$, k
 - $V = V'$
 - $E = E'$
 - $(G = G')$
 - $k = |V'| - k'$
 - Esta transformação é polinomial:
 - Tempo constante para construir $G = (V, E)$
 - $O(|V|)$ para contar o número de vértices
 - Provar que há um V_I ($|V_I| = k'$) para G' se, e se somente, há um V_C ($|V_C| = k$) para G .

Vertex Cover é NP-completo?

- (4) Provar que G' tem um conjunto independente V_I de tamanho k' se, e se somente, VC tem uma cobertura V_C de tamanho k .
 - Considere dois conjuntos I e J | $I \cap J = \emptyset$, $I \cup J = V = V'$
 - Dada qualquer aresta (u, v) , um dos seguintes casos se verifica:
 1. $u, v \in I$
 2. $u \in I$ e $v \in J$
 3. $u \in J$ e $v \in I$
 4. $u, v \in J$

Vertex Cover é NP-completo?

- (4) Continuação...
 - Assumindo-se que I é um conjunto independente de G' , então:
 - O caso 1 não pode ser (vértices em I não podem ser adjacentes)
 - Nos casos 2 e 3, (u, v) tem *exatamente um* ponto terminal em J
 - No caso 4, (u, v) tem *ambos* os pontos terminais em J
 - Nos casos 2, 3 e 4, (u, v) tem *pelo menos um* ponto terminal em J
 - Então, vértices em J cobrem todas as arestas de G'
 - Também: $|I| = |V| - |J|$ uma vez que $I \cap J = \emptyset$, $I \cup J = V = V'$
 - Assim, se I é um conjunto independente de G' , então J é uma cobertura dos vértices de G' ($= G$)
 - Similarmente, pode-se provar que se J é uma cobertura dos vértices de G' , então I é um conjunto independente de G'

Exemplos de problemas *NP*-completo

- Alguns exemplos são
 - Ciclos de Hamilton
 - Coloração em grafos
 - Cliques em grafos
 - Subgrafos e supergrafos
 - Árvores de expansão
 - Cortes e conectividade
 - Problemas de fluxo
 - Outros...

Referências e mais informação

- T. Cormen *et al.* (2009) “**Introduction to Algorithms.**” Cambridge, MA: MIT press.
- R. Johnsonbaugh & M. Schaefer (2004) “**Algorithms.**” Upper Saddle River, NJ: Prentice Hall.
- C.A. Shaffer (2001) “**A Practical Introduction to Data Structures and Algorithm Analysis.**” Upper Saddle River, NJ: Prentice Hall.

Exames



Mestrado Integrado em Engenharia Informática e Computação

Complementos de Programação e Algoritmos

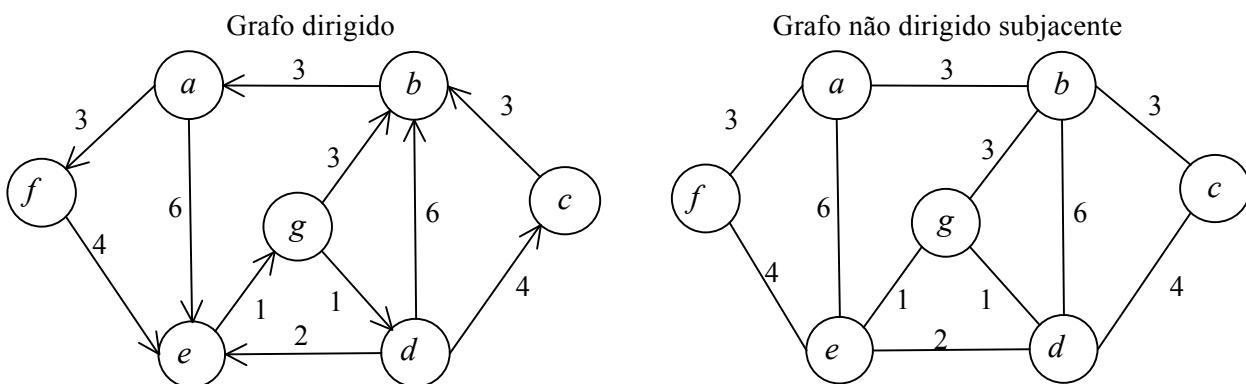
EXAME COM CONSULTA

11 Junho de 2007

DURAÇÃO: 2 horas

- 1.** [6] Relativamente ao grafo dirigido ou ao grafo não dirigido subjacente da figura seguinte indique se existe e, em caso de existir, apresente:

- a) [1] um circuito de Euler (no grafo dirigido);
- b) [1] um circuito de Hamilton (no grafo dirigido);
- c) [1] um fluxo de valor ≥ 6 de f (fonte) para c (poço), supondo que os pesos representam capacidades e que o fluxo pode passar nas arestas em qualquer sentido (no grafo não dirigido);
- d) [1] uma árvore de expansão de peso ≤ 14 (no grafo não dirigido);
- e) [1] um percurso do carteiro chinês de peso ≤ 40 (no grafo não dirigido);
- f) [1] um emparelhamento de peso ≥ 13 (no grafo não dirigido).

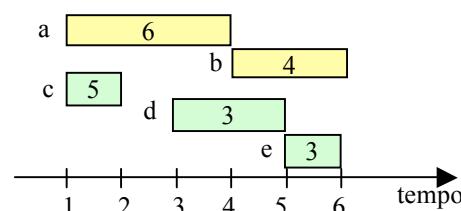


- 2.** [6] Dado um conjunto de n actividades com instantes de início e fim $[s_i, f_i]$ e peso w_i , para $i = 1, \dots, n$, pretende-se achar um subconjunto S de actividades não sobrepostas de peso total máximo.

Considere o seguinte algoritmo ganancioso: inserir em S a actividade de peso máximo w_i , remover todas as actividades que se sobreponem à actividade i e repetir até não restarem actividades. Na prática, não é necessário remover actividades, basta processá-las por pesos decrescentes e inserir em S as que não se sobreponem a outras já em S .

Por exemplo, no caso das actividades representadas por rectângulos na figura ao lado, o algoritmo ganancioso escolheria as actividades $\{a, b\}$, de peso total 10.

Note-se que o algoritmo ganancioso nem sempre garante a solução óptima, que neste caso seria $\{c, d, e\}$, de peso total 11.



- a) [5] Implemente em Java este algoritmo ganancioso num método estático público `selectNonOverlapping` da classe `Activity`, partindo do seguinte esqueleto (pode criar métodos auxiliares e alterar a classe como melhor entender e deve usar colecções do Java):

```
import java.util.Set;
public class Activity {
    private long start, finish, weight;
    // Construtor e selectores omitidos
    public static Set<Activity> selectNonOverlapping(Set<Activity> s) {...}
}
```

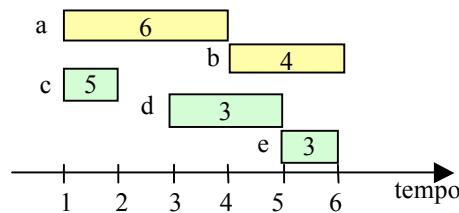
Nota: se não se recordar do nome e comportamento exacto de algum método das bibliotecas do Java, indique em comentário o comportamento esperado.

- b) [1] Indique, justificando, a complexidade espacial e temporal do programa.



Os exercícios seguintes analisam 3 estratégias alternativas para encontrar uma solução óptima para o problema proposto no exercício 2.

Repete-se ao lado o exemplo para facilitar a leitura.



3. [4] Uma solução óptima para o problema 2 pode ser encontrada eficientemente com base na técnica de programação dinâmica.

Inicialmente, ordenam-se as actividades por forma a definir os arrays:

- `activity[i]` - i-ésima actividade por ordem de instantes de fim crescentes;
- `prev[i]` - maior índice j tal que $j < i$ e as actividades i e j não se sobrepõem, isto é, tal que $f_j \leq s_i$ (é 0 se não existir nenhum j nessas condições).

De seguida aplica-se a técnica de programação dinâmica preenchendo dois arrays:

- `best[i]` - custo da melhor solução que se consegue usando apenas as actividades de 1 a i ;
- `incl[i]` - indica se a actividade i está incluída na melhor solução com actividades de 1 a i .

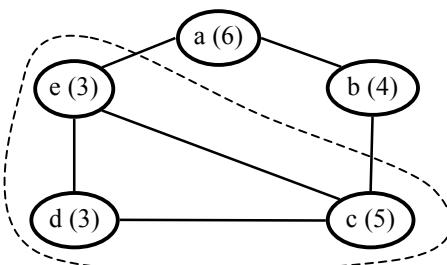
A tabela seguinte exemplifica o preenchimento destes arrays para o exemplo da figura acima.

<code>i</code>	0	1	2	3	4	$n = 5$
<code>activity[i]</code>	-	c	a	d	e	b
<code>prev[i]</code>	-	0	0	1	3	2
<code>best[i]</code>	0	5	6	8	11	11
<code>incl[i]</code>	-	true	true	true	true	false

- [1] Indique por código Java as fórmulas e ordem de cálculo dos arrays `best[i]` e `incl[i]`.
- [1] Indique por código Java o processo de obtenção da solução óptima S (do tipo `Set<Activity>`) a partir dos arrays já calculados.
- [1] Indique (por palavras) como se pode calcular o array `prev[i]` em tempo $O(n \log n)$.
- [1] Indique, justificando, a complexidade espacial e temporal do algoritmo.

4. [2] Uma solução óptima para o problema 2 pode ser encontrada eficientemente através da sua redução ao problema de encontrar um caminho simples de peso máximo (caminho "mais comprido") num grafo dirigido acíclico, o qual pode ser resolvido em tempo linear no tamanho do grafo. Explique como se pode efectuar essa redução, exemplificando para o caso da figura (mostrar o grafo correspondente e o caminho óptimo).

5. [2] Uma solução óptima para o problema 2 pode ser encontrada de forma potencialmente ineficiente através da sua redução ao problema de encontrar um clique (subgrafo completo) de peso máximo num grafo não dirigido com pesos nos vértices. Para esse efeito, cria-se um grafo em que os vértices representam as actividades (com os respectivos pesos) e as arestas ligam actividades não sobrepostas. A figura ao lado mostra o grafo e a solução óptima para o exemplo dado.

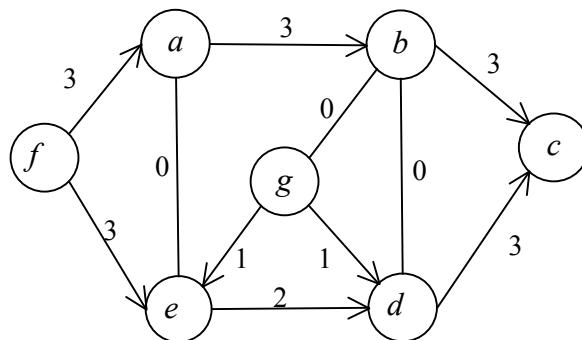


Sabendo-se que o problema de encontrar um clique de tamanho máximo num grafo não dirigido é NP-completo, mostre que o problema de encontrar um clique de peso máximo num grafo não dirigido com pesos nos vértices é também NP-completo. Sendo assim, justifique porque é que esta estratégia de resolução do problema 2 é potencialmente ineficiente.

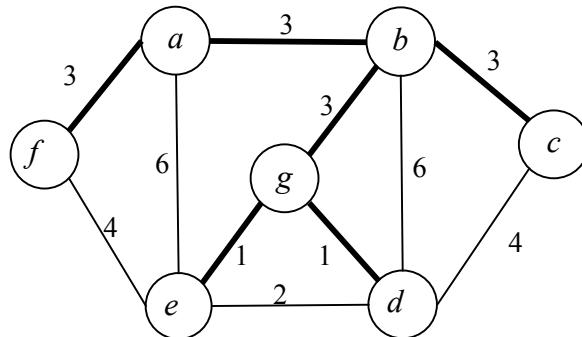
FIM

**Resolução****1.**

- a) Não tem nenhum circuito de Euler porque existem vértices com diferentes graus de entrada (nº de arestas a entrar) e saída (nº de arestas a sair), como por exemplo o vértice a .
- b) Sim - o circuito: $f \rightarrow e \rightarrow g \rightarrow d \rightarrow c \rightarrow b \rightarrow a \rightarrow f$.
- c) Sim. É possível passar um fluxo de valor 6, como indica o seguinte grafo de fluxos:

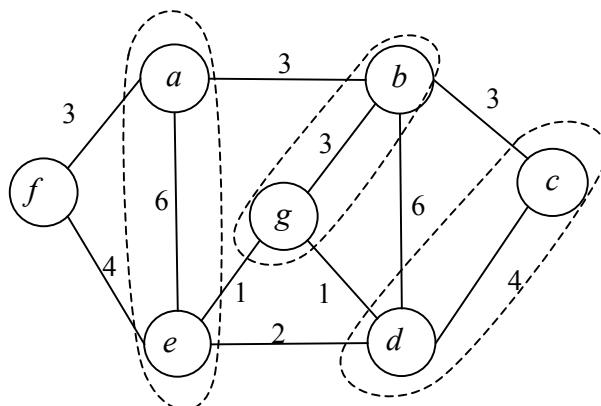


- d) Sim, pois a árvore de expansão mínima (indicada na figura seguinte) tem precisamente peso 14.



- e) Começa-se por achar o percurso óptimo (de peso mínimo) do carteiro chinês. Os únicos vértices de grau ímpar são os vértices a e g e o caminho mais curto entre eles é o caminho $a-b-g$, que é necessário duplicar para se obter um grafo Euleriano. Somando os pesos das arestas do grafo original (36) com o peso das arestas duplicadas (6), obtemos $42 > 40$. Logo a resposta é não.

- f) Sim. O emparelhamento $\{(a,e), (b,g), (d,c)\}$ (ver figura seguinte) tem peso total 13.





2.

a) Uma resolução possível:

```
import java.util.Set;
import java.util.HashSet;
import java.util.PriorityQueue;

public class Activity implements Comparable<Activity> {
    private long start, finish, weight;

    // Não pedido, criado só para efectuar testes
    public Activity(long start, long finish, long weight) {
        assert start <= finish && weight >= 0;
        this.start = start; this.finish = finish; this.weight = weight;
    }

    // Obtém um subconjunto de actividades não sobrepostas de peso máximo
    public static Set<Activity> selectNonOverlapping(Set<Activity> s) {
        PriorityQueue<Activity> q = new PriorityQueue<Activity>(s);
        Set<Activity> result = new HashSet<Activity>();
        while ( ! q.isEmpty() ) {
            // nota: extraí mínimo da fila que, dada a definição de compareTo,
            // vai dar a actividade de peso máximo
            Activity a = q.poll();
            if ( ! a.overlapAny(result) )
                result.add(a);
        }
        return result;
    }

    // Verifica se esta actividade se sobrepõe a alguma dum conjunto dado
    private boolean overlapAny(Set<Activity> s) {
        for (Activity a : s)
            if ( this.overlap(a) )
                return true;
        return false;
    }

    // Verifica se esta actividade (this) se sobrepõe a outra (a)
    public boolean overlap(Activity a) {
        return this.start < a.finish && a.start < this.finish;
    }

    // Compara esta actividade (this) com outra (a) por peso de forma inversa
    public int compareTo(Activity a) {
        return Long.signum(a.weight - this.weight);
        // signum dá +1, 0 ou -1 conforme o argumento é >0, 0 ou <0
    }
}
```

Nota: Não seria boa ideia tentar usar a capacidade de eliminação de duplicados de HashSet, pois seria necessário definir não só o método equals mas também o método hashCode (por forma a retornarem true e o mesmo valor de hash para acontecimentos duplicados).

b) Nesta implementação (pouco optimizada) a complexidade temporal é $T(n) = O(n^2 + n \log n) = O(n^2)$. A parcela $O(n \log n)$ tem a ver com $2n$ operações na fila de prioridades. A parcela $O(n^2)$ tem a ver com n execuções de overlapAny. A complexidade espacial é $S(n) = O(n)$ devido à utilização da fila de prioridades.



3. a) best[0] = 0;
for (int i = 1; i <= n; i++) {
 best[i] = Math.max(best[i-1], activity[i].weight + best[prev[i]]);
 incl[i] = best[i] > best[i-1];
}

b) for (int i = n; i > 0;)
 if (incl[i]) {
 result.add(activity[i]);
 i = prev[i];
 }
 else
 i--;

c) Ordena-se o array startorder[] = {1, 2, ..., n}, em que os nºs identificam actividades no array activity[], por ordem de instantes de início crescentes. Percorrendo em paralelo os arrays startorder[] e activity[] consegue-se calcular o array prev[]:

```
for (int i = 0, j = 0; j < n; j++) {  
    while (i < n && activity[i+1].finish <= activity[startorder[j]].start)  
        i++;  
    prev[startorder[j]] = i;  
}
```

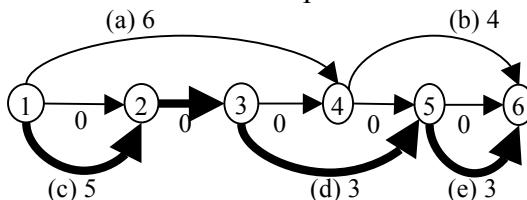
Dado que a ordenação pode ser feita em tempo $O(n \log n)$ e o ciclo acima corre em tempo $O(n)$ (a instrução `i++` é executada n vezes no máximo), `prev[]` é calculado em tempo $O(n \log n)$.

d) A construção e ordenação do array `activity[]` pode ser feita em tempo $O(n \log n)$. De acordo com a alínea anterior, o array `prev[]` pode ser calculado no mesmo tempo. O resto do trabalho (alíneas a) e b)) é feito em tempo linear. Logo a complexidade temporal é $T(n) = O(n \log n)$. A complexidade espacial é obviamente $O(n)$.

4. Cria-se um grafo em que os vértices representam os instantes de tempo referenciados nas várias actividades (instantes de início ou fim, sem repetições). Criam-se arestas de peso 0 a ligar esses vértices por ordem cronológica. Representa-se cada actividade por uma aresta dirigida do respectivo instante de início para o respectivo instante de fim, com o respectivo peso.

Nota 1: Usando um HashSet, pode-se construir o conjunto de instantes de tempo / vértices, sem repetições, em tempo $O(n)$. A ordenação necessária para adicionar as arestas de peso 0 pode ser feita em tempo $O(n \log n)$. As arestas correspondentes às actividades podem ser adicionadas em tempo $O(n)$. Uma vez que o tamanho do grafo é de ordem $O(n)$, o caminho óptimo é encontrado em tempo $O(n)$. Tem-se assim a mesma complexidade espacial e temporal que no exercício 3! Em alternativa, podia-se construir um grafo de precedências entre actividades, mas a complexidade espacial e temporal piorava para $O(n^2)$.

Exemplo:



5. Reduz-se o problema do clique de tamanho máximo num grafo G ao problema do clique de peso máximo num grafo G', considerando pesos unitários. Por outro lado, é óbvio que o problema do clique de peso máximo está em NP (uma solução pode ser verificada em tempo polinomial). Assim, o problema do clique de peso máximo é NP-completo. Isto implica que, segundo o estado actual do conhecimento, não existe nenhum algoritmo de tempo polinomial para resolver o problema do clique de peso máximo. Portanto é má ideia usar esta abordagem para resolver o problema 2.



Mestrado Integrado em Engenharia Informática e Computação

Complementos de Programação e Algoritmos

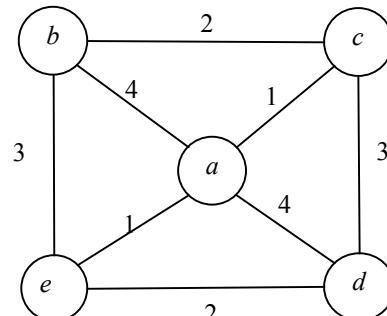
EXAME COM CONSULTA

9 Julho de 2007

DURAÇÃO: 2 horas

- 1.** [5] Relativamente ao grafo da figura ao lado indique se existe e, em caso de existir, apresente:

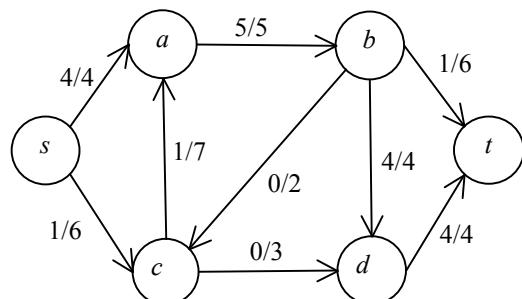
- a) [1] um conjunto de no máximo 3 vértices cobrindo todas as arestas;
- b) [1] um clique de tamanho ≥ 4 ;
- c) [1] um percurso do caixeiro viajante de peso ≤ 12 (percurso fechado que passa pelo menos uma vez em cada vértice);
- d) [2] um percurso do carteiro chinês de peso ≤ 24 (percurso fechado que passa pelo menos uma vez em cada aresta).



Nota: na alínea d), começar por encontrar um percurso óptimo do carteiro chinês, seguindo os passos estudados nas aulas.

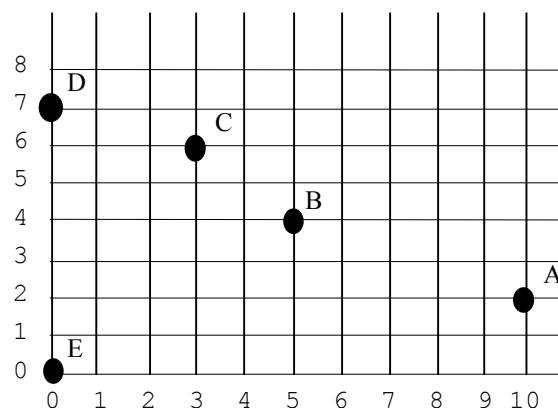
- 2.** [2.5] Considerar o grafo de fluxos e capacidades numa rede de transporte indicado na figura ao lado.

- a) [0.5] Qual é o valor do fluxo que passa na rede?
- b) [1] Desenhe o grafo de resíduos correspondente.
- c) [0.5] Identifique um caminho de aumento no grafo de resíduos e o valor de fluxo correspondente.
- d) [0.5] Actualize o grafo de fluxos aplicando o caminho de aumento anterior.



- 3.** [2.5] Considere cinco cidades localizadas nas coordenadas A(10,2), B(5,4), C(3,6), D(0,7) e E(0,0). Pretende-se ligar estas cidades com uma quantidade mínima de cabo. As distâncias Euclidianas aproximadas entre estas cidades estão representadas no quadro abaixo.

- a) [0.5] Que algoritmo usaria para resolver o problema?
- b) [0.5] Desenhe o grafo a fornecer como entrada para o algoritmo.
- c) [1.5] Obtenha uma solução para o problema usando o algoritmo. Indicar todos os passos.



	B	C	D	E
A	5.4	8.1	11.2	10.2
B		2.8	5.8	6.4
C			3.2	6.7
D				7.0



4. [2] Considere as strings S="ABCD" e T="ACED".

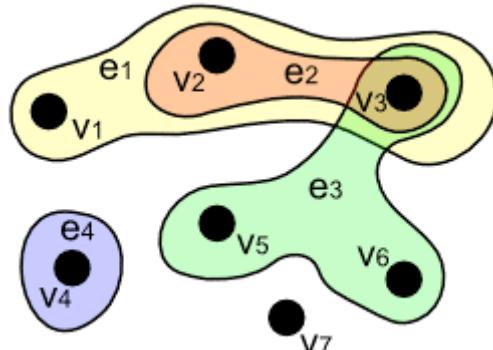
a) [0.5] Qual é a distância de edição entre estas strings?

b) [1.5] Construa a matriz D[i,j] estudada nas aulas para determinar a distância de edição pelo método de programação dinâmica.

5. [3] Um hipergrafo é uma generalização de um grafo em que as arestas podem ligar qualquer número de vértices. Formalmente, um hipergrafo é um par (V,E) em que V é um conjunto de elementos, chamados *nós* ou *vértices*, e E é um conjunto de subconjuntos não vazios de X chamados *hiperarestas*. A figura ao lado mostra um exemplo de um hipergrafo.

a) [1] Mostre que o problema de verificar se um hipergrafo tem um conjunto com K ou menos vértices cobrindo todas as arestas é NP-completo.

b) [2] Indique (por palavras) um algoritmo de aproximação ganancioso para obter um conjunto de vértices de tamanho mínimo cobrindo todas as arestas. Que resultado daria no caso da figura (em que a solução óptima é $\{v_3, v_4\}$). Qual é a eficiência espacial e temporal desse algoritmo?



$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \\ E &= \{e_1, e_2, e_3, e_4\} \\ &= \{\{v_1, v_2, v_3\}, \{v_2, v_3\}, \{v_3, v_5, v_6\}, \{v_4\}\}. \end{aligned}$$

6. [5] Escreva uma classe Hypergraph em Java que permita executar o código de teste indicado abaixo, referente ao exemplo do problema 5. Implemente o algoritmo ganancioso do problema 5.b) no método getMinVertexCover.

```
class HypergraphTest extends TestCase {
    public void testExample() {
        Hypergraph g = new Hypergraph();

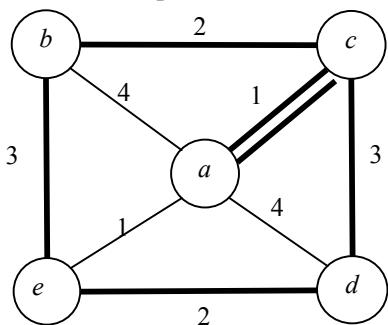
        g.addVertex(1); g.addVertex(2); g.addVertex(3); g.addVertex(4);
        g.addVertex(5); g.addVertex(6); g.addVertex(7);

        g.addEdge(1); g.addEdgeToEdge(1, 1); g.addEdgeToEdge(2, 1);
                    g.addEdgeToEdge(3, 1);
        g.addEdge(2); g.addEdgeToEdge(2, 2); g.addEdgeToEdge(3, 2);
        g.addEdge(3); g.addEdgeToEdge(3, 3); g.addEdgeToEdge(5, 3);
                    g.addEdgeToEdge(6, 3);
        g.addEdge(4); g.addEdgeToEdge(4, 4);

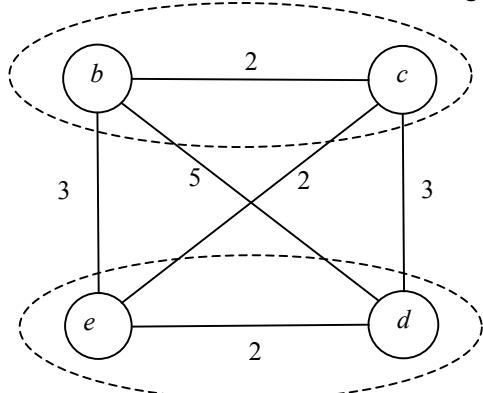
        Set<Integer> vertexCover = new HashSet<Integer>();
        vertexCover.add(3); vertexCover.add(4);

        assertEquals( vertexCover, g.getMinVertexCover());
    }
}
```

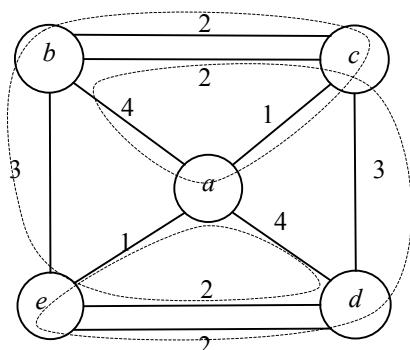
Nota: A parte da classe Hypergraph que permite representar e construir o hipergrafo vale 2.5 valores; a parte referente ao método getMinVertexCover vale os restantes 2.5 valores.

**Resolução****1.****a)** Existe: {a, b, d} ou {a, c, e}**b)** Não existe. Só existem cliques de tamanho 3 (qualquer dos triângulos do grafo).**c)** Existe. Por exemplo o percurso que passa nas arestas indicadas a traço forte na figura passa em todos os vértices e tem peso total 12.**d)**

Grafo com distâncias entre vértices de grau ímpar, e emparelhamento óptimo de peso mínimo:



Grafo inicial duplicando caminhos mais curtos entre vértices emparelhados, e um percurso possível (peso total 24).

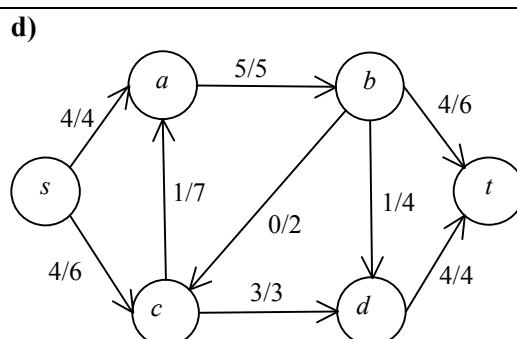
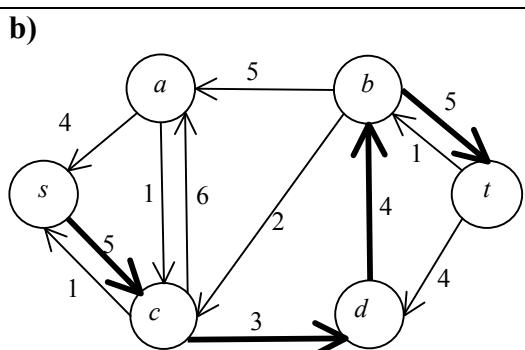


Como o peso total deste percurso é 24, a resposta é afirmativa.



2.

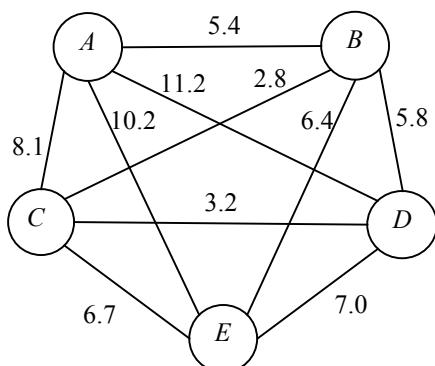
a) 5



c) Ver caminho a traço forte na figura anterior.
O valor do fluxo de aumento é 3.

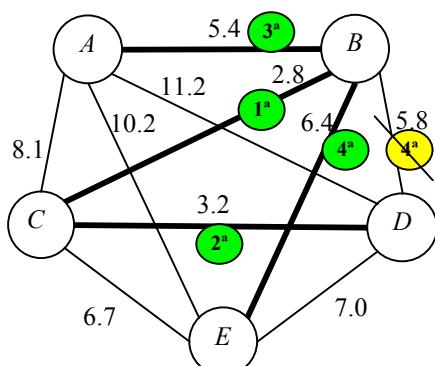
3. a) Determinação da árvore de expansão mínima, por exemplo pelo algoritmo de Kruskal.

b)

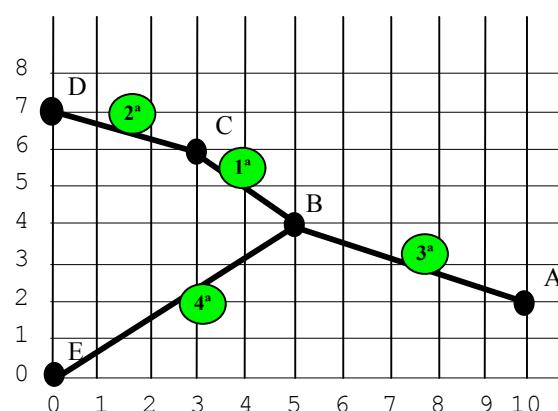


c) Escolhe-se sucessivamente uma aresta de peso mínimo que não causa um ciclo, até ligar todos os vértices.

Resultado e ordem de construção
(arestas a traço forte e nºs de ordem):



Visto agora no "mapa" (não pedido):





4. a) A distância de edição é 2 (por exemplo, em T pode-se substituir C por B e E por C).

b)

D[i,j]		A	C	E	D
	0	1	2	3	4
A	1	0	1	2	3
B	2	1	1	2	3
C	3	2	1	2	3
D	4	3	2	2	2

5. a) É necessário provar que:

- (i) está em NP (i.e., que uma solução potencial pode ser verificada em tempo polinomial);
- (ii) existe um problema NP-completo que é redutível a este.
- (i) Dada uma potencial solução W (subconjunto de vértices), é trivial verificar em tempo polinomial que $|W| \leq K$ e que todas as hiperarestas são cobertas pelos vértices em W.
- (ii) Sabe-se que o problema da cobertura de vértices em grafos normais (em que cada aresta liga dois vértices) é NP-completo. Um grafo normal pode ser visto como um hipergrafo em que por acaso todas as hiperarestas incidem em 2 vértices (ou pode ser trivialmente convertido para um hipergrafo desse tipo).

b) (i) Escolhe-se o vértice que cobre mais hiperarestas, simula-se a eliminação desse vértice e das hiperarestas por ele cobertas, e repete-se o processo até não existem mais hiperarestas.

(ii) No exemplo da figura seria escolhido primeiro o vértice v_3 e depois o vértice v_4 , dando portanto a solução óptima $\{v_3, v_4\}$.

(iii) *Eficiência temporal* (detalhando o algoritmo): Seja n o nº de vértices, m o nº de hiperarestas e r o nº de pares (vértice, aresta). Para evitar a eliminação de vértices e hiperarestas, mantém-se em cada vértice um contador do nº de hiperarestas remanescentes cobertas por esse vértice, e mantém-se em cada aresta uma flag a indicar se já foi coberta. No início o contador é inicializado com o nº de hiperarestas incidentes no vértice e a flag é inicializada com false, o que pode ser feito em tempo $O(n+m)$. Em cada iteração, o vértice com o maior valor do contador pode ser escolhido em tempo $O(n)$. A simulação da eliminação consiste em percorrer as arestas incidentes no vértice seleccionado e, para cada aresta que não estava ainda coberta (flag=false), marcar a flag a true e decrementar os contadores dos vértices abrangidos por essa aresta. As várias simulações de eliminação podem ser feitas em tempo $O(r)$ se existirem listas ligadas que permitem aceder dos vértices às arestas e vice-versa. O tempo total do algoritmo fica então $O(n^2+r)$. Usando uma fila com prioridades para guardar os vértices com o de valor mais alto do contador à cabeça, consegue-se $O(r \log n + r)$, que pode ser melhor que o anterior, dependendo do valor de r .

Eficiência espacial: se não for usada uma fila de prioridades, é necessário apenas o espaço para guardar os dados auxiliares (contador e flag), ou seja, $O(n+m)$.

6.

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.Map;
import java.util.Set;

public class Hypergraph {
    static class Vertex {
        int num;
        LinkedList<Edge> edges = new LinkedList<Edge>();
        int numRemEdges; // nº de arestas cobertas (para getMinVertexCover)
        public Vertex(int num) {
            this.num = num;
        }
    }
}
```



```
static class Edge {  
    int num;  
    LinkedList<Vertex> vertices = new LinkedList<Vertex>();  
    boolean covered; // usado por getMinVertexCover  
    public Edge(int num) {  
        this.num = num;  
    }  
}  
  
// Conjuntos de vértices e arestas acessíveis pelo seu número  
private Map<Integer,Vertex> vertexSet = new HashMap<Integer,Vertex>();  
private Map<Integer,Edge> edgeSet = new HashMap<Integer, Edge>();  
  
public void addVertex(int num) {  
    vertexSet.put(num, new Vertex(num));  
}  
public void addEdge(int num) {  
    edgeSet.put(num, new Edge(num));  
}  
public void addVertexToEdge(int vertexNum, int edgeNum) {  
    Edge e = edgeSet.get(edgeNum);  
    Vertex v = vertexSet.get(vertexNum);  
    v.edges.add(e);  
    e.vertices.add(v);  
}  
  
public Set<Integer> getMinVertexCover() {  
    // Inicializa resultado e dados auxiliares  
    Set<Integer> result = new HashSet<Integer>();  
    int numCovered = 0;  
    for (Vertex v : vertexSet.values())  
        v.numRemEdges = v.edges.size();  
    for (Edge e : edgeSet.values())  
        e.covered = false;  
    // Ciclo principal  
    while (numCovered != edgeSet.size()) {  
        // Escolhe o vértice que cobre mais arestas remanescentes  
        // Nota: podia ser optimizado com fila de prioridades  
        Vertex bestV = null;  
        int numRemEdges = 0;  
        for (Vertex v : vertexSet.values()) {  
            if (v.numRemEdges > numRemEdges) {  
                numRemEdges = v.numRemEdges;  
                bestV = v;  
            }  
        }  
        // Acrescenta esse vértice ao resultado e actualiza dados auxiliares  
        for (Edge e: bestV.edges)  
            if ( ! e.covered ) {  
                result.add(bestV.num);  
                numCovered++;  
                e.covered = true;  
                for (Vertex w : e.vertices)  
                    w.numRemEdges --;  
            }  
    }  
    return result;  
}
```

Complementos de Programação e Algoritmos

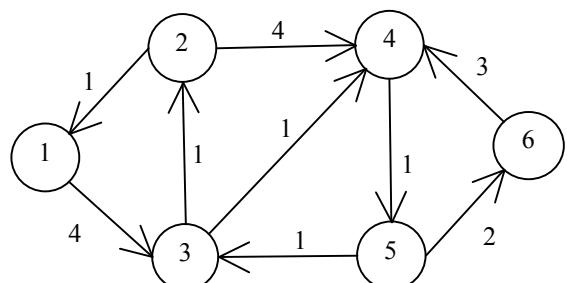
EXAME COM CONSULTA

11 Junho de 2008

DURAÇÃO: 2 horas

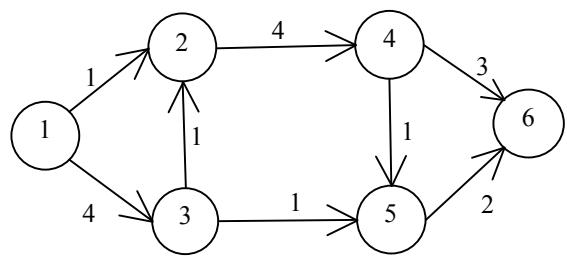
1. [7] Relativamente ao grafo dirigido da figura ao lado:

- a) [1] indique se existe e, em caso de existir, apresente uma ordenação topológica dos vértices do grafo;
- b) [1] indique um caminho mais curto (de peso total mínimo) do vértice 5 para o vértice 1;
- c) [1] indique se o grafo é conexo ou fortemente conexo, justificando.



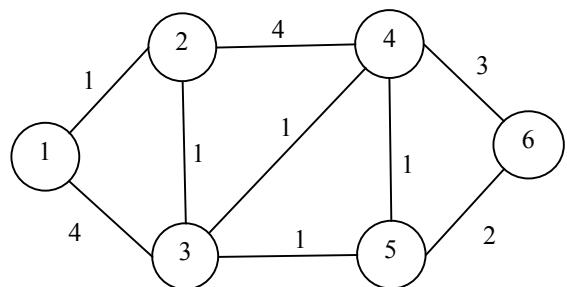
Relativamente à rede de transporte da figura ao lado

- d) [1] indique o fluxo máximo que pode passar entre o nó 1 e o nó 6 (assumir que os pesos nas arestas indicam capacidades e apresentar o mesmo grafo com o fluxo que passa em cada aresta).



Relativamente ao grafo não dirigido da figura ao lado indique, justificando:

- e) [1] uma árvore de expansão mínima;
- f) [1] um caminho de Euler entre os vértices 2 e 5;
- g) [1] todos os pontos de articulação existentes.



2. [7] Um condutor pretende efectuar uma viagem longa seguindo um percurso predefinido, ao longo do qual se encontram diversas bombas de abastecimento de combustível que praticam preços variados, e pretende planejar os abastecimentos a efectuar de forma a minimizar o valor gasto em abastecimentos. Os abastecimentos devem ser planeados por forma a que a quantidade de combustível não desça abaixo de um nível mínimo de segurança.

Os dados de entrada para o problema são:

- m - capacidade máxima do depósito, em litros;
- q - quantidade de combustível existente inicialmente no depósito, em litros;
- s - quantidade mínima de segurança de combustível no depósito, em litros;
- c - consumo de combustível em litros/km (suposto constante);
- $\{(d_i, p_i)\}, \dots, (d_n, p_n)\}$ - para cada bomba (i) existente ao longo do percurso, a distância ao ponto de partida em km (d_i) e o preço unitário praticado em euros/litro (p_i);
- d - distância a percorrer.

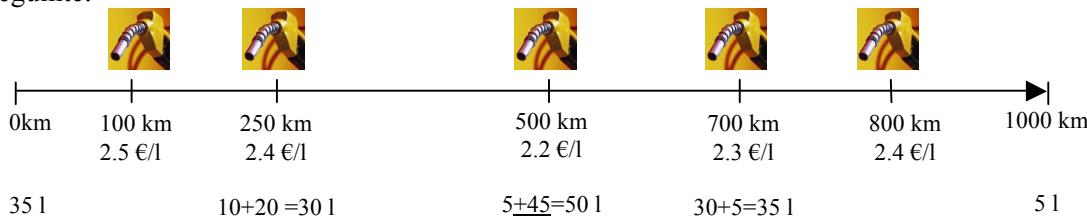
Os dados de saída pretendidas são:

- q_1, \dots, q_n - para cada bomba (i) existente ao longo do percurso, a quantidade (q_i) de combustível a abastecer em litros (0 no caso de não abastecer).

Por exemplo, dados os seguintes dados de entrada:

- $m = 50\text{ l}$ $q = 35\text{ l}$ $s = 5\text{ l}$ $c = 0,1\text{ l/km}$ $d = 1000\text{ km}$
- $\{(d_i, p_i)\} = \{(100\text{km}, 2.5\text{€/l}), (250\text{km}, 2.4\text{€/l}), (500\text{km}, 2.2\text{€/l}), (700\text{km}, 2.3\text{€/l}), (800\text{km}, 2.4\text{€/l})\}$

o plano de abastecimento óptimo (e quantidade de combustível) ao longo do percurso é esquematizado na figura seguinte:



- a) [4] Conceba um algoritmo eficiente para obter o plano de abastecimento óptimo. Explique o algoritmo genericamente e apresente-o em pseudo código baseado em Java.
- b) [1] Indique, justificando, que técnica(s) de concepção de algoritmos aplicou neste caso.
- c) [1] Aplique o algoritmo passo a passo ao exemplo dado.
- d) [1] Indique, justificando, a complexidade temporal do algoritmo. Em particular, indique que estruturas de dados e colecções do Java seriam usadas na implementação para conseguir a eficiência anunciada.

3. [6] Pretende-se determinar o caminho mais curto a seguir entre dois pontos numa rede viária, com a restrição de garantir que há pontos de abastecimento suficientes ao longo do percurso. Isto é, ao longo do percurso escolhido, a distância entre o ponto de partida e a primeira bomba de abastecimento, entre bombas consecutivas, e entre a última bomba e o ponto de chegada, não pode exceder a autonomia do veículo.

- a) [4] Com base nos algoritmos em grafos estudados, conceba um algoritmo o mais eficiente possível para determinar o melhor caminho a seguir. É dado o mapa de estradas com as distâncias entre nós e a localização das bombas de abastecimento de combustível, bem como os pontos de partida e chegada no mapa. Apresentar os passos principais do algoritmo, explicando cada passo por palavras.
- b) [2] Indique, justificando, a eficiência temporal e espacial do algoritmo.

Tópicos de resolução

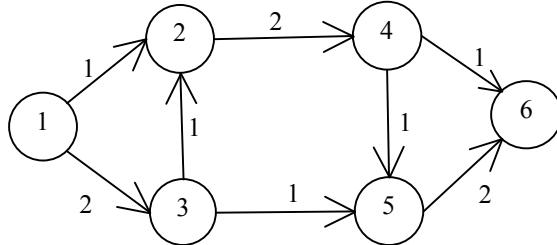
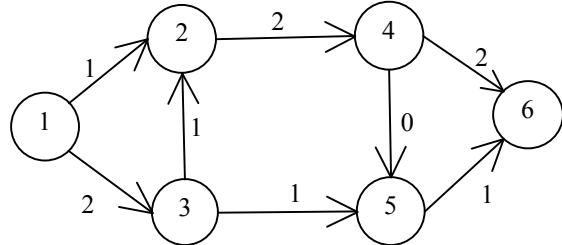
1.

a) Não existe uma ordenação tipológica dos vértices, porque o grafo tem ciclos ($1 \rightarrow 3 \rightarrow 2 \rightarrow 1$, etc.).

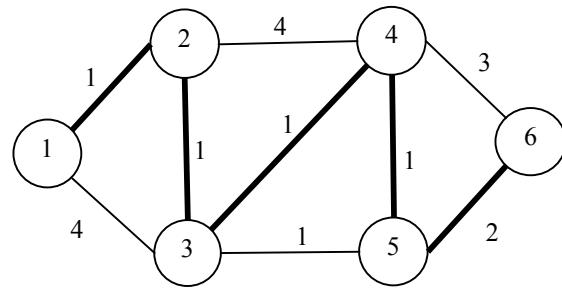
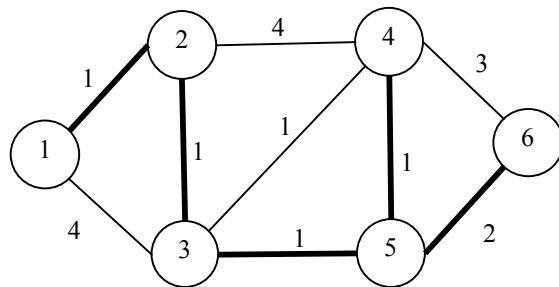
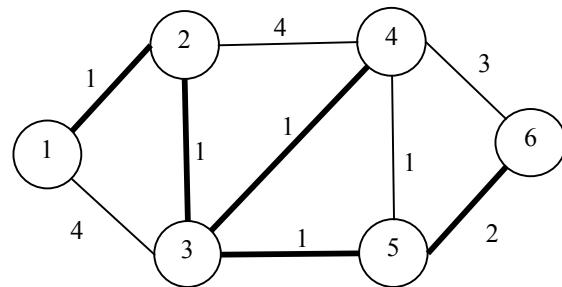
b) $5 \rightarrow 3 \rightarrow 2 \rightarrow 1$ (peso 3).

c) É fortemente conexo, pois existe um caminho dirigido entre quaisquer dois vértices.

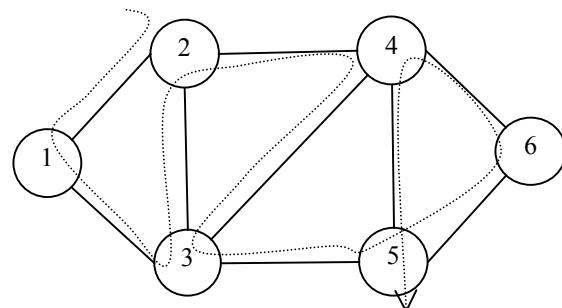
d) Fluxo máximo é 3. Grafo de fluxos pode ser um dos seguintes:



e) Soluções possíveis (indicar apenas uma):



f) Existe porque os vértices inicial e final têm grau ímpar e todos os outros têm grau par. Um caminho possível é: $2 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 5$, conforme ilustrado na figura (existem mais possibilidades).



g) Não tem pontos de articulação.

2. a)

Solução 1 - Algoritmo ganancioso

1. Começar no início do percurso

$d_{corrente} = 0; q_{corrente} = q; q_i = 0 \ (i=1, \dots, n);$

2. Se combustível for suficiente p/ chegar ao fim (acima nível segurança), não é preciso abastecer (termina)

$d_{max} = d_{corrente} + (q_{corrente} - s) / c;$

$\text{if } (d_{max} \geq d) \text{ return true;}$

3. Escolhe a bomba alcançável com preço mais baixo

$i = \text{select } i \in \{1, \dots, n\} : p_i \text{ is min} \wedge d_i \in [d_{corrente}, d_{max}]$

4. Se não existir nenhuma bomba alcançável, falha

$\text{if } (i == \text{null}) \text{ return false;}$

5. Avança até à bomba escolhida

$q_{corrente} -= (d_i - d_{corrente}) * c; d_{corrente} = d_i;$

6. Se o destino for alcançável a partir desta bomba, abastece só o necessário para chegar ao destino

$d_{max} = d_i + (m - s) / c;$

$\text{if } (d_{max} \geq d) \{ q_i = (d - d_i) * c + s - q_{corrente}; q_{corrente} += q_i; \}$

7. Senão, se existe uma bomba mais barata alcançável a partir desta, abastece só o necessário para chegar lá

$\text{else if } (\text{exists } j \in \{1, \dots, n\} : p_j < p_i \wedge d_j \in [d_i, d_{max}])$

$\{ q_i = (d_j - d_i) * c + s - q_{corrente}; q_{corrente} += q_i; \}$

8. Senão, atesta

$\text{else } \{ q_i = m - q_{corrente}; q_{corrente} = m; \}$

9. Continua no passo 2

$\text{goto } 2$

Solução 2 - Algoritmo de divisão e conquista

1. Começar por considerar todo o percurso

$d_{ini} = 0; d_{fim} = d; q_{ini} = q; i_{min} = 1; i_{max} = n; q_i = 0 \ (i=1, \dots, n);$

2. Se combustível for suficiente p/ chegar ao fim (acima nível segurança), não é preciso abastecer (termina)

$d_{max} = d_{ini} + (q_{ini} - s) / c;$

$\text{if } (d_{max} \geq d) \text{ return true;}$

3. Se não existir nenhuma bomba no percurso, falha

$\text{if } (i_{min} > i_{max}) \text{ return false;}$

4. Escolhe a bomba com preço mais baixo no percurso

$i = \text{select } i \in \{i_{min}, \dots, i_{max}\} : p_i \text{ is min};$

5. Abastece o máximo possível nessa bomba (chegando com mínimo e saindo com máximo)

$q_{chegada} = \max(s, q_{ini} - (d_i - d_{ini}) * c);$

$q_{saída} = \min(m, (d_{fim} - d_i) * c + s);$

$q_i = q_{saída} - q_{chegada};$

6. Repete o procedimento (passos 2 a 7) do início até esta bomba

$\text{repetir com } d_{fim} = d_i; i_{max} = i-1;$

7. Repete o procedimento (passos 2 a 7) desta bomba até ao fim

$\text{repetir com } d_{ini} = d_i; i_{min} = i+1; q_{ini} = q_{saída};$

c) Mostrar o que se passa em cada iteração.

d) $O(n^2)$ - no máximo n (ou $2n$) iterações, cada uma $O(n)$ (procurar bomba mais barata)

3. a) Construir um grafo com as distâncias mais curtas entre todos os pares de bombas, bem como entre cada bomba e o início e fim (cada aresta representa um caminho no mapa original com uma distância). Eliminar arestas de distância superior à autonomia do veículo. Achar depois o caminho mais curto do ponto de partida ao ponto de chegada.

Complementos de Programação e Algoritmos

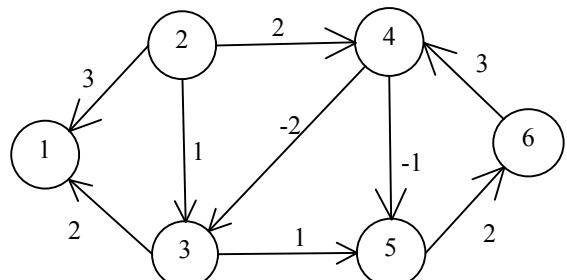
EXAME COM CONSULTA

7 Julho de 2008

DURAÇÃO: 2 horas

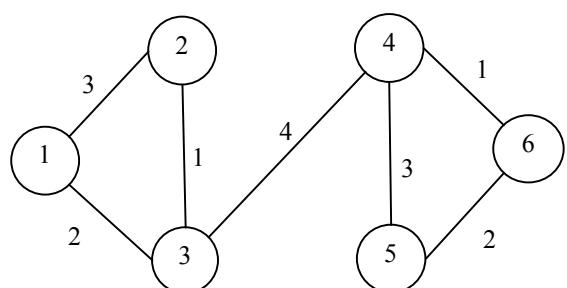
1. [5] Relativamente ao grafo dirigido da figura ao lado:

- a) [2] obtenha o caminho mais curto entre o vértice 2 e o vértice 6, aplicando passo a passo um algoritmo adequado estudado na disciplina;
- b) [1] indique se o grafo é conexo ou fortemente conexo, justificando.



Relativamente ao grafo não dirigido da figura ao lado indique, justificando:

- c) [1] a quantidade mínima de cabo para ligar todos os vértices (assumindo que os pesos indicam distâncias);
- d) [1] todos os pontos de articulação existentes.



2. [3] Considere a frase “este teste é chato”

- a) [1] Proponha justificando um código de tamanho fixo para codificar a frase e diga quantos bits precisa.
- b) [2] Se utilizar uma codificação de Huffman, quantos bits precisaria? Mostre os passos de construção da árvore que utilizou e quais os códigos associados a cada símbolo.

3. [6] Na prescrição electrónica de medicamentos, coloca-se o problema de distribuir o conjunto de medicamentos prescritos pelo médico para um paciente, por um número mínimo de impressos de prescrição (vulgo receita médica), atendendo a que cada impresso só aceita um número limitado de medicamentos e podem existir restrições que impedem misturar certos medicamentos no mesmo impresso.

Por exemplo, se o médico prescrever os medicamentos ABCDE, cada impresso só aceitar 3 medicamentos, e os pares de medicamentos CE, DE e BE não poderem ficar no mesmo impresso, uma solução possível é repartir os medicamentos por 2 impressos da seguinte forma: AE e BCD.

- a) [3] Conceba um algoritmo eficiente (capaz de correr em tempo polinomial) para efectuar a repartição dos medicamentos pelos impressos de prescrição, procurando minimizar o número de impressos de prescrição. Apresente o algoritmo por uma sequência de passos numerados e escritos em linguagem natural (português), auxiliado por expressões matemáticas ou pseudo-código que julgar adequado para retirar ambiguidade. Indique, justificando, que técnica(s) de concepção de algoritmos aplicou neste caso.
- b) [1] Ilustre a aplicação do algoritmo passo a passo ao exemplo dado.
- c) [1] Indique, justificando, a complexidade temporal e espacial do algoritmo.
- d) [1] Indique, justificando, se o algoritmo garante a solução óptima (com número mínimo de impressos). No caso negativo, indique um contra-exemplo e explique que outro tipo de algoritmo se poderia usar para encontrar a solução óptima.

- 4. [6]** Pretende-se construir uma infra-estrutura pública (hospital, posto de bombeiros, etc.) para servir um conjunto de cidades.

Essa infra-estrutura deve ser localizada na cidade mais "central", no sentido de estar o mais perto possível da cidade mais distante.

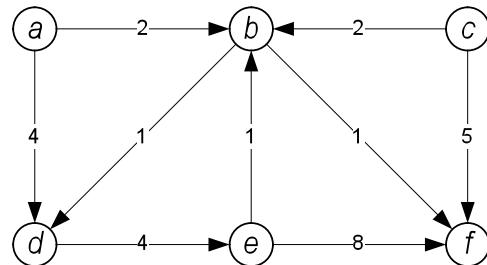
Por exemplo, relativamente às cidades indicadas no mapa ao lado, a cidade de Viseu é a mais central (a uma distância de 250 km das cidades mais distantes - Viana do Castelo e Bragança).



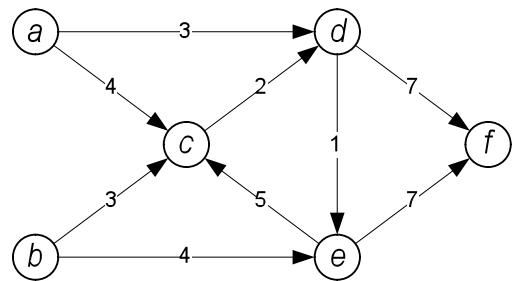
- [3] Com base nos algoritmos em grafos estudados, conceba um algoritmo eficiente para determinar a cidade mais central. É dado um mapa de estradas com as cidades e distâncias, semelhante ao apresentado na figura. Apresente o algoritmo por uma sequência de passos numerados e descritos em linguagem natural (português), auxiliado eventualmente por expressões matemáticas ou pseudo-código para retirar ambiguidade.
- [1] Ilustrar a aplicação do algoritmo para o exemplo dado.
- [1] Indique, justificando, a eficiência temporal e espacial do algoritmo.
- [1] Suponha agora que é necessário assegurar uma certa distância máxima em relação a qualquer das cidades a servir, instalando a infra-estrutura em mais do que uma cidade se necessário. Por exemplo, se quisermos instalar urgências de pediatria que distam no máximo 100km de qualquer cidade do mapa acima, poder-se-iam instalar urgências em Guarda, Aveiro, Braga e Bragança. Esboce e indique a eficiência de um algoritmo para resolver este problema (isto é, seleccionar um conjunto mínimo de cidades de forma a que todas as outras distem no máximo uma certa distância de uma cidade seleccionada).

Mestrado Integrado em Engenharia Informática e Computação	Complementos de Programação e Algoritmos	EXAME COM CONSULTA	17 Junho de 2009	DURAÇÃO: 2 horas
---	---	--------------------	-------------------------	------------------

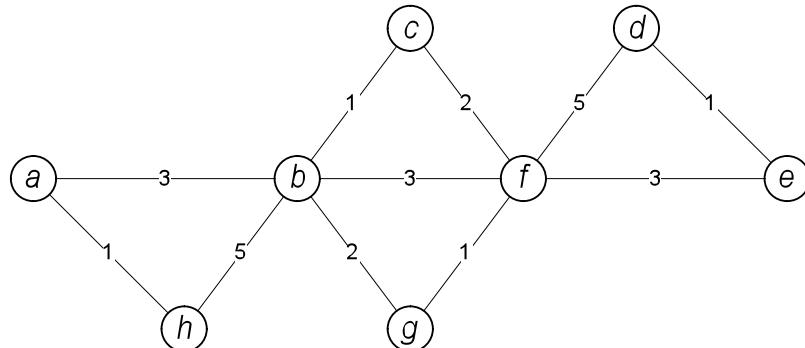
- 1.** [4] Relativamente ao grafo dirigido da figura ao lado:
- [1] Indique uma ordenação topológica dos vértices do grafo;
 - [1] Se o peso da aresta eb fosse -1 , não haveria ordenação topológica possível. Comente;
 - [1] Indique o caminho de peso total mínimo entre os vértices a e f ;
 - [1] Classifique o grafo como não conexo, conexo ou fortemente conexo, justificando a sua opção.



- 2.** [1] Considere a rede de distribuição de água representada pelo grafo da figura ao lado:
- [1,5] Considerando que os nós a e b são ambos fontes, e que os números nos arcos da rede correspondem às capacidades máximas da tubagem respectiva, indique o fluxo máximo que se pode esperar no nó f .
 - [0,5] Do ponto de vista do fluxo máximo, o que acontecerá se uma das fontes (a ou b) falhar?



- 3.** [4] Considere o grafo não dirigido abaixo e responda às questões seguintes, mostrando todos os passos que efectuou para chegar à solução.



- [1] Indique uma árvore de expansão mínima.
- [1] Se aplicar os algoritmos de *Prim* e *Kruskal* na alínea anterior, pode-se obter diferentes árvores de expansão mínima? Justifique.
- [1] Indique, se existir, um caminho de *Euler* entre os vértices b e f .
- [1] Indique todos os pontos de articulação existentes.

4. [5] Imagine que tem uma balança com dois pratos e um conjunto P de n pesos com valores p_1, p_2, \dots, p_n . Pretende-se distribuir os pesos pelos dois pratos da balança de forma a ficar equilibrada. Como não existe a certeza se tal solução existe dentro do conjunto P , o objectivo é minimizar a diferença de peso entre os pratos.

- a) [3] Concea um algoritmo eficiente (capaz de correr em tempo polinomial) que distribua os pesos pelos dois pratos da balança, minimizando a diferença de peso. Apresente o algoritmo por uma sequência de passos numerados e escritos em linguagem natural (português), auxiliado por expressões matemáticas ou pseudo-código (ou Java, se preferir) que julgar adequado para retirar ambiguidade. Indique, justificando, que técnica(s) de concepção de algoritmos aplicou neste caso.
- b) [1] Indique, justificando, a complexidade temporal e espacial do algoritmo.
- c) [1] Indique, justificando, se o algoritmo garante a solução óptima. No caso negativo, que outro tipo de algoritmo se poderia usar para encontrar a solução óptima.

5. [5] Um ladrão profissional acabou de assaltar um museu. Planeou a sua fuga através da rede de metro, em que pretende sair e entrar em todas as estações do seu percurso de fuga para tentar despistar a Polícia que o persegue. Para tal, tenta optimizar o seu percurso escolhendo as estações com mais pessoas para passar mais despercebido.

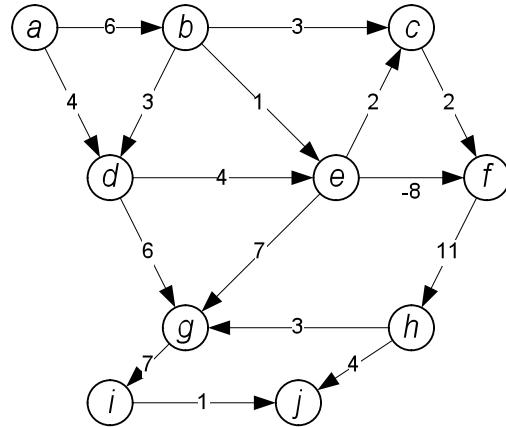
- a) [4] Com base nos algoritmos em grafos estudados, conceba um algoritmo o mais eficiente possível para determinar o melhor percurso de fuga a seguir. É dado o mapa do metro com as médias de ocupação de pessoas em cada estação, bem como as estações de partida e chegada. Apresentar os passos principais do algoritmo, explicando cada passo por palavras. Indique igualmente de que forma construiria o grafo de entrada.
- b) [1] Indique, justificando, a eficiência temporal e espacial do algoritmo.

Boa Sorte

Mestrado Integrado em Engenharia Informática e Computação
Complementos de Programação e Algoritmos (Época de Recurso)
 EXAME COM CONSULTA **13 Julho de 2009** DURAÇÃO: 2 horas

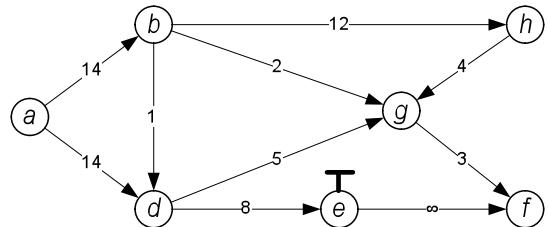
1. [2] Relativamente ao grafo dirigido da figura ao lado, responda justificando:

- a) [1] Indique uma ordenação topológica dos vértices do grafo, se houver;
- b) [1] Indique o caminho de peso total mínimo entre os vértices **a** e **j**;

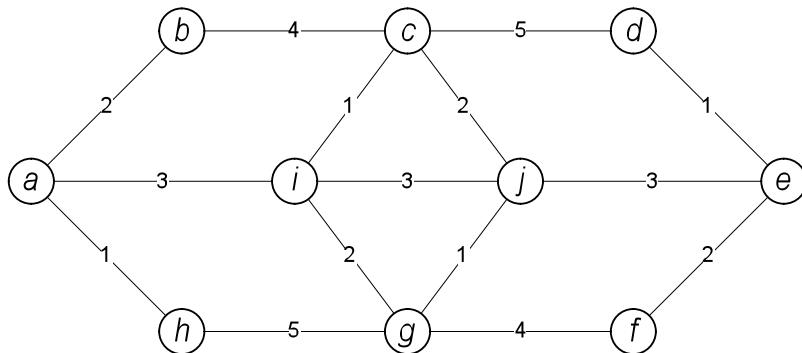


2. [2 valores] Considere a rede de distribuição de água representada pelo grafo da figura ao lado. Considere também que os números nos arcos da rede correspondem às capacidades máximas da tubagem respectiva, o nó **a** é fonte, os nós **f** e **h** são ambos drenos, e o nó **e** possui uma válvula controlável de capacidade máxima de 10 unidades de fluxo. Responda às questões seguintes justificando a sua resposta:

- a) [1] Qual o máximo fluxo que se pode esperar no nós **f** e **h**, conjuntamente?
- b) [1] Caso a válvula do nó **e** seja fechada pela metade, que efeito será observado no fluxo dos nós **f** e **h**?



3. [4] Considere o grafo não dirigido abaixo e responda às questões seguintes, mostrando todos os passos que efectuou para chegar à solução ou justificando a sua resposta.



- a) [1] Indique uma árvore de expansão mínima, utilizando o algoritmo de **Prim**.
- b) [1] Ao multiplicarmos o peso da aresta **i-j** por (-1), o algoritmo de **Kruskal** deixará de funcionar? Explique.
- c) [1] Indique um caminho de **Euler** entre os vértices **a** e **e**, se existir.
- d) [1] Indique todos os pontos de articulação existentes.

Nas questões 4, 5 e 6, apresente o algoritmo desenvolvido por uma sequência de passos numerados e escritos em linguagem natural (português), auxiliado por expressões matemáticas ou pseudo-código (ou Java, se preferir) que julgar adequado para retirar quaisquer ambiguidades.

4. [3 valores] É-lhe pedido para colocar um conjunto de n crianças em fila india, assim como também lhe é fornecida uma lista de m declarações do tipo “criança i não gosta da criança j ”. Se i não gosta de j , então não é deseável que i fique em alguma posição atrás de j ; caso contrário, i seria capaz de atirar qualquer coisa à cabeça de j e provocar zaragata. Construa um algoritmo que ordene a fila desejada em tempo $O(m + n)$, ou justifique caso não seja possível.

5. [3 valores] Conceba um algoritmo de tempo de execução linear que recebe como entrada um grafo dirigido acíclico $G = (V, E)$ e dois vértices, s e t pertencentes a V , e retorna o número de caminhos entre s e t em G . O algoritmo precisa apenas contar e devolver o número de caminhos, e não listá-los.

6. [6 valores] Um sistema de planeamento de viagens para utentes de transportes públicos pretende oferecer um serviço que permita planear viagens multi-modais, ou seja, definindo-se o ponto de origem e de destino da viagem, o sistema é capaz de sugerir, de acordo com dadas restrições, a melhor combinação de meios de transportes, incluindo autocarros, metro, ou percursos a pé. Considere que as redes de autocarro e metro são conhecidas, assim como a localização das respectivas paragens. O custo de viagem é equivalente à soma dos tempos necessários para transpor cada arco da rede. Considere também que as pessoas movem-se ao logo dos eixos da rede viária (rede de autocarros e carros), nos passeios e que todas as velocidades médias são conhecidas (para autocarros, metro e pessoas a pé).

- a) [5]** Com base na teoria dos grafos conceba um algoritmo eficiente para determinar o melhor plano de viagem que combine os transportes necessários para minimizar o tempo de percurso a pé, seguido da minimização do tempo de viagem em transportes.
- b) [1]** Comente a eficiência temporal e espacial do algoritmo concebido.

ATENÇÃO! Entregue o exame em duas partes separadas, a primeira contendo as questões 1, 2 e 3, e a segunda contendo as questões 3, 4 e 5.

Boa Sorte!

Mestrado Integrado em Engenharia Informática, MIEIC

Concepção e Análise de Algoritmos, CAL (2009-2010)

Exame com Consulta

22 de Junho de 2010

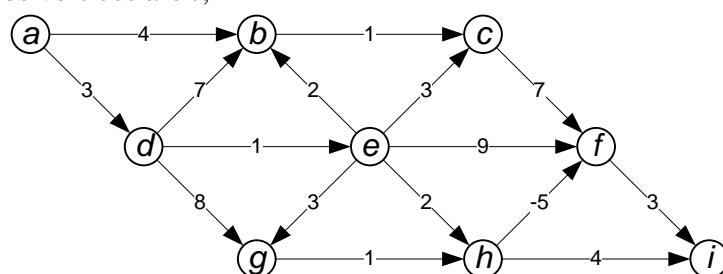
Duração: 2 horas

Nome: _____

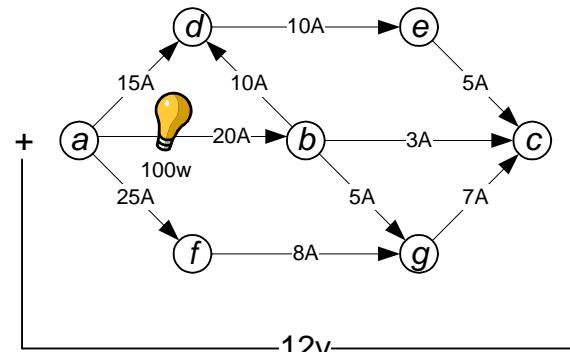
Número: _____

1. (4 valores) Considere o grafo dirigido da figura abaixo e responda às alíneas seguintes:

- [1] Classifique o grafo quanto à sua conectividade, justificando a sua resposta;
- [1] Indique uma ordenação topológica possível, se existir;
- [1] Se invertermos o sentido da aresta (e, c) , o número de ordenações topológicas aumentará? Comente e justifique;
- [1] Considerando que os pesos das arestas são distâncias, indique o caminho de distância mínima entre os vértices a e i ;



2. (4 valores) A potência eléctrica (**P**, medida em Watt) é uma grandeza que relaciona duas das unidades básicas da electricidade, nomeadamente a tensão (**V**, medida em Volt) e a corrente (**I**, medida em Ampere), da seguinte forma: $P = V * I$. Considere que o circuito eléctrico da figura abaixo está ligado a uma fonte de tensão de 12 Volt e que uma lâmpada de 100 Watt está ligada ao circuito como se indica na figura. Os valores das arestas indicam a corrente máxima da cablagem utilizada nessa aresta.

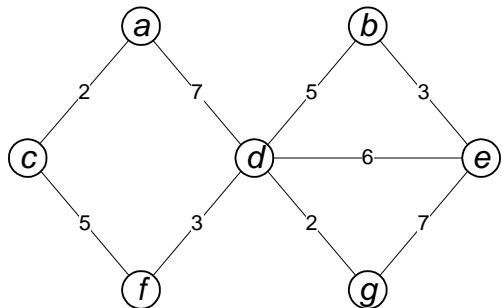


- [2] Indique qual a corrente eléctrica máxima que se pode esperar no nó c deste circuito;
 - [1] A lâmpada utilizada está sub ou sobredimensionada para o circuito em questão? Justifique;
 - [1] O que acontecerá se houver um curto-círcuito entre os nós e e c do circuito? Justifique.
3. (4 valores) Considere todos os caracteres da seguinte frase: "Portugal marcou sete golos contra a Coreia do Norte!", incluindo espaços em branco e não considerando se as letras são maiúsculas e minúsculas.
- [1] Proponha um código de tamanho fixo para codificar a expressão, justificando a sua opção e indicando quantos bits precisa;
 - [2] Utilize, demonstrando passo a passo, a codificação de Huffman para a mesma expressão;
 - [1] Compare, justificando, as eficiências temporais e espaciais do código de Huffman e do código sugerido por si na alínea a);

V.S.F.F.

4. (4 valores) Considere o grafo pesado, da figura abaixo e responda às alíneas seguintes:

- [1] Indique todos os pontos de articulação do grafo, caso existam. Justifique a sua resposta;
- [1] Comente a afirmação “o grafo tem um circuito de Euler, mas não um caminho de Euler”. Justifique;
- [2] Encontre um “Caminho do Carteiro Chinês”, a começar no vértice a . Explique, passo a passo, o método utilizado para o cálculo deste caminho.



5. (4 valores) O Porto de Leixões é a maior infra-estrutura portuária do Norte de Portugal, por onde são enviados produtos para exportação através dos vários navios que acedem ao porto diariamente. Antes do embarque, os produtos chegam ao porto dentro de diversos contentores de dimensões e capacidades (m^3) padronizadas, e são agrupados de acordo com as suas capacidades, $C = \{c_1, c_2, \dots, c_i, \dots, c_{x-1}, c_x\}$. Por outro lado, os navios que transportam esses contentores também têm capacidade de carga limitada a um determinado volume (m^3) máximo, que define a sua categoria, $N = \{n_1, n_2, \dots, n_i, \dots, n_{y-1}, n_y\}$. Durante um dia típico, o sistema de carga do porto apenas tem capacidade para carregar um número limitado de navios, z , que são atendidos por ordem de chegada ao porto.

- [2,5] Concea um algoritmo eficiente, capaz de carregar o maior número possível de contentores e maximizar a utilização da capacidade do navio a ser carregado. Apresente o algoritmo por uma sequência de passos numerados e escritos em linguagem natural (português), auxiliado por expressões matemáticas ou pseudo-código (ou Java, se preferir) que julgar adequado para retirar ambiguidade.
- [0,5] Indique, justificando, que técnica(s) de concepção de algoritmos aplicou neste caso e se o algoritmo garante a solução óptima. No caso negativo, que outro tipo de algoritmo se poderia usar para encontrar a solução óptima?
- [1] Comente, justificando, a complexidade temporal e espacial do algoritmo.

IMPORTANTE!

- O enunciado deve obrigatoriamente ser entregue com as folhas de resposta, e identificado com o nome e número do aluno;
- Responda às questões em folhas duplas, distribuindo-as da seguinte forma:

Questões 1 e 2

Questões 3 e 4

Questão 5

Bom Exame!

Mestrado Integrado em Engenharia Informática, MIEIC

Concepção e Análise de Algoritmos, CAL (2009-2010)

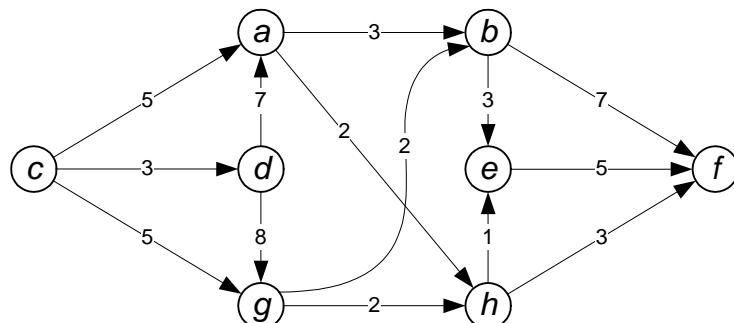
Exame com Consulta

16 de Julho de 2010

Duração: 2 horas

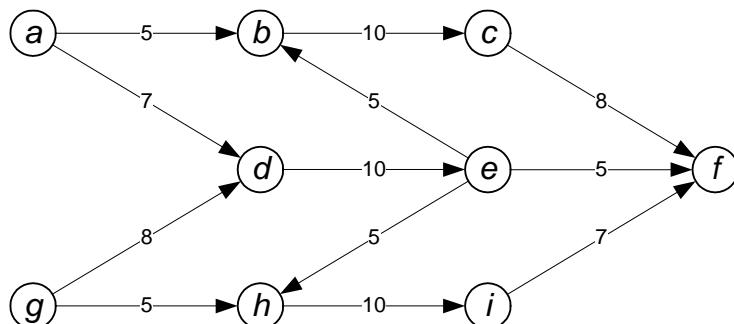
Nome: _____

1. (4 Valores) Considere a rede de auto-estradas mantidas pela Brisa, representada no grafo dirigido da figura ao lado. Os pesos das arestas representam o custo anual de manutenção de cada troço da rede. Responda às questões abaixo, justificando e/ou apresentando todos os passos dos cálculos que efectuou.



- a) [1] Indique o caminho de menor custo de manutenção, entre os vértices *c* e *f*.
- b) [1] Indique o caminho de maior custo de manutenção, entre os vértices *c* e *f*.
- c) [1] Indique uma ordenação topológica, se houver.
- d) [1] Comente a seguinte afirmação: “Ao invertermos o sentido da aresta (*a*, *h*), deixará de haver ordenação topológica possível”.

2. (4 Valores) O grafo dirigido da figura ao lado representa a rede de distribuição de gás numa zona urbana, onde os pesos das arestas indicam a capacidade da tubulação. Responda às questões abaixo, justificando e/ou apresentando todos os passos dos cálculos que efectuou.



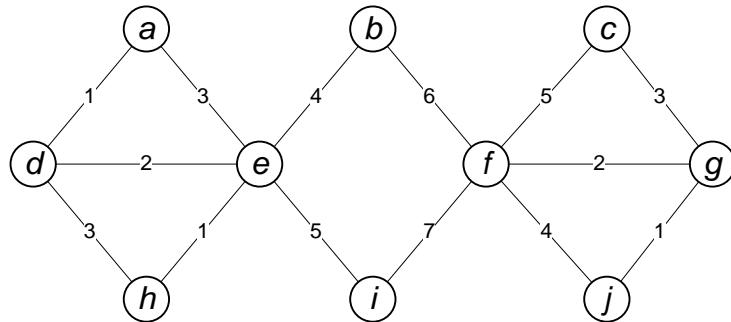
- a) [2] Qual o fluxo total que se pode esperar no vértice *f*?
- b) [1] Se houver uma fuga de gás na aresta (*d*, *e*), o que acontece ao fluxo da rede?
- c) [1] Se houver uma obstrução na aresta (*e*, *f*), o que acontece ao fluxo da rede?

3. (4 Valores) Considere a string “oficioso”, na qual se pretende realizar uma pesquisa aproximada do padrão “facial”, e responda às questões seguintes:

- a) [2] Construa a matriz de programação dinâmica para este problema.
- b) [1] Indique a distância de edição.
- c) [0,5] Comente as condições de fronteira da matriz gerada.
- d) [0,5] Comente a optimização espacial e temporal que se pode obter ao utilizarmos outra estrutura de dados.

V.S.F.F.

4. (4 Valores) Considere o grafo não dirigido abaixo e responda às questões seguintes, mostrando todos os passos que efectuou para chegar à solução ou justificando a sua resposta:



- a) [1,5] Indique uma árvore de expansão mínima, utilizando o algoritmo de *Kruskal*.
 - b) [1] Comente a seguinte afirmação: “Ao retirarmos o vértice *a*, o algoritmo de *Prim* deixará de funcionar.”
 - c) [1] Indique um caminho de *Euler* entre os vértices *d* e *g*, se existir.
 - d) [0,5] Indique todos os pontos de articulação existentes, se houver.
5. (4 Valores) Atente ao problema dos coeficientes binomiais (também conhecido por “Triângulo de Pascal”). Para as seguintes expressões:

$$(x+y)^2 = x^2 + 2xy + y^2, \text{ os coeficientes são } 1,2,1.$$

$$(x+y)^3 = x^3 + 3x^2y + 3xy^2 + y^3, \text{ os coeficientes são } 1,3,3,1.$$

$$(x+y)^4 = x^4 + 4x^3y + 6x^2y^2 + 4xy^3 + y^4, \text{ os coeficientes são } 1,4,6,4,1.$$

Os $n-1$ coeficientes podem ser calculados para $(x+y)^n$ segundo a fórmula:

$$C(n,i) = \frac{n!}{i!(n-i)!}, \text{ para } 0 \leq i < n$$

- a) [1] Escreva a matriz de coeficientes para $C(n,i)$ para $n=6$.
- b) [2] Apresente, usando Java, uma solução óptima para este problema usando programação dinâmica. A função em causa deverá retornar o valor do coeficiente de índice **m** num binómio de grau **n**, e deverá ter a seguinte assinatura:

```
public static int binom(int n, int m).
```

- c) [1] Indique e justifique a complexidade espacial e temporal do algoritmo.

IMPORTANTE!

1. O enunciado deve obrigatoriamente ser entregue com as folhas de resposta, e identificado com o nome e número do aluno;
2. Responda às questões em folhas duplas, distribuindo-as da seguinte forma:

Questões 1 e 2

Questões 3 e 4

Questão 5

Bom Exame!

Mestrado Integrado em Engenharia Informática, MIEIC

Concepção e Análise de Algoritmos, CAL (2009-2010)

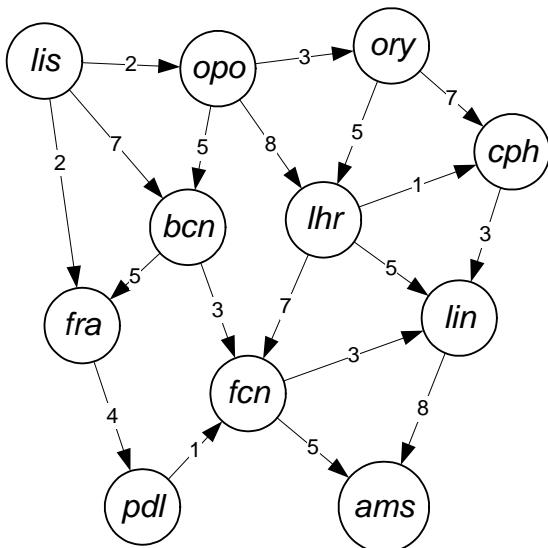
Exame com Consulta

26 de Julho de 2010

Duração: 2 horas

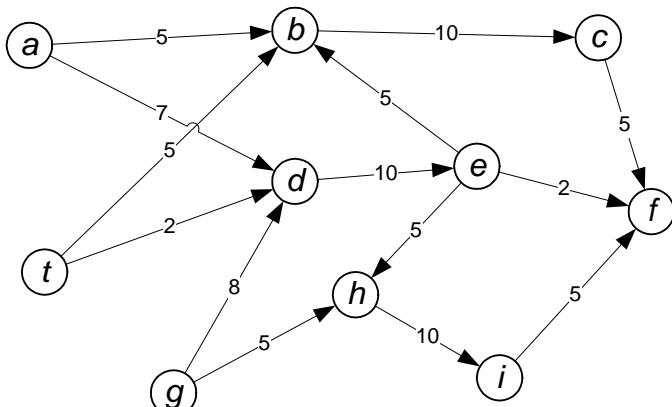
Nome: _____

1. (4 Valores) Considere a rede de voos da companhia aérea **RecursoVaiàVida**, representada no grafo dirigido da figura ao lado. Os vértices representam aeroportos e as arestas representam as ligações existentes entre esses aeroportos. Os pesos das arestas representam o número de ligações. Responda às questões abaixo, justificando:



- Indique duas ordenações topológicas, caso existam.
- Indique um caminho correspondente ao maior número de ligações bem como o maior número de ligações possível, entre os aeroportos de Lisboa (*lis*) e de Amsterdão (*ams*).
- A companhia aérea contratou um gestor de topo chamado **ComigoOuVaiOuRacha** e tomou a decisão de aumentar o número de ligações entre *lhr* e *lin* para 10. Qual é o impacto dessa decisão no maior número possível de ligações entre *lis* e *ams*? E nos caminhos correspondentes?
- A companhia aérea chegou à conclusão que o gestor anterior não correspondia às necessidades e, por isso, contratou um outro chamado **ComigoIstoéPraCair**. A primeira decisão do novo gestor foi de anular as 10 ligações entre *lhr* e *lin*, substituindo-as por 5 ligações entre *lin* e *lhr*. Qual é o impacto desta decisão no maior número possível de ligações entre *lis* e *ams*? E na topologia do grafo?

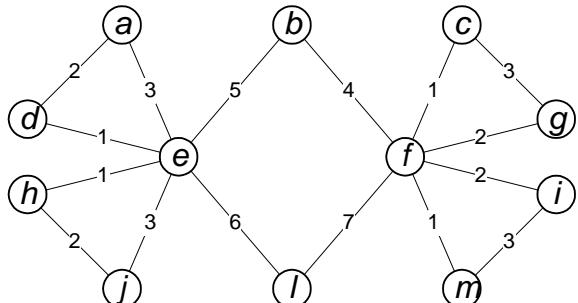
2. (4 Valores) Os **Umpa Lumpa** são homens pequeninos que trabalham na fábrica de chocolate de **Willy Wonka** e que fazem a ligação entre as várias estações de trabalho da fábrica. O grafo dirigido da figura ao lado representa a rede de produção da fábrica. Os vértices são estações de trabalho dos **DetestoUmpaLumpas** e as arestas representam os túneis por onde os **Umpa Lumpa** passam a correr. Os pesos das arestas indicam a capacidade máxima dos túneis em *umpalumpas per min*. Responda às questões abaixo, justificando:



- Qual o fluxo total que se pode esperar na estação de trabalho *f*?
- Um número não determinado de **Umpa Lumpas** tropeçou no túnel correspondente à aresta (*d, e*) e, por isso, não se consegue garantir a capacidade máxima desse túnel. Indique os vários cenários possíveis relativamente ao que pode acontecer ao fluxo da rede.
- O túnel correspondente à aresta (*e, f*) é a subir com uma inclinação de 20%. Um **Umpa Lumpa** mais radical e discordante com as condições de trabalho, resolveu dinamitar o túnel. O que acontece ao fluxo da rede?

3. (4 Valores) Considere todos os caracteres da expressão “**Rússia não bebia Coca-cola. Agora, até Pepsi Cola!**”, incluindo espaços em branco e não considerando acentos ou se as letras são maiúsculas ou minúsculas:

- a) [1] Proponha um código de tamanho fixo para codificar a expressão, justificando a sua opção e indicando quantos bits precisa;
- b) [2] Utilize, demonstrando passo a passo, a codificação de Huffman para a mesma expressão;
- c) [1] Comente e compare os diversos tipos de codificação apresentados nas aulas, aplicados à expressão acima, no que refere às eficiências temporais e espaciais e sugira a melhor opção, justificando a sua resposta.
4. (4 Valores) Considere o grafo pesado, da figura ao lado, e responda às alíneas seguintes justificando e/ou apresentando todos os passos dos cálculos que efectuou:
- a) [1] Indique um “caminho de Euler” possível, caso exista;
- b) [1] Indique um “circuito de Euler” possível, caso exista;
- c) [1] Encontre um “Caminho do Carteiro Chinês”, a começar no vértice a , caso exista;
- d) [1] Caso existam, indique todos os pontos de articulação do grafo, demonstrando e/ou justificando a sua resposta.



5. (4 Valores) Imagine que na sua quinta no **Farmville**, pretende fazer uma colheita. No entanto, o seu tractor não tem combustível suficiente para fazer toda a colheita. Assim, apenas é possível percorrer o terreno numa única passagem, começando no canto superior esquerdo e terminando no canto inferior direito. Como o terreno tem plantações variadas, o lucro em cada cela varia, segundo um determinado valor apresentado na figura ao lado. Visto que o tractor não pode voltar para trás, correndo o risco de ficar sem combustível, isso implica que o movimento no terreno se tenha de fazer apenas movendo uma posição de cada vez, da esquerda para a direita ou de cima para baixo. Usando programação dinâmica, escreva uma solução óptima na qual se obtenha o valor máximo de lucro, respeitando estas restrições. A fórmula de recorrência para o cálculo das soluções intermédias é:

$$S[i,j] = A[i,j] + \max \begin{cases} S[i-1,j], & \text{se } i > 0 \\ S[i,j-1], & \text{se } j > 0 \\ 0, & \text{nos outros casos} \end{cases}$$

6	4	6	3	7	8	3	8
2	7	3	6	6	3	7	5
16	9	6	3	4	5	1	3
7	4	9	12	11	9	3	13
7	4	7	5	3	2	7	5
2	9	18	6	6	9	8	8
9	6	2	13	8	9	1	2
15	8	6	4	2	3	7	5
6	1	4	2	7	9	5	5

Onde $A[i,j]$ é o valor do lucro disponível na cela (i,j) . Basta portanto construir a matriz S , de soluções intermédias, obtendo-se a melhor solução na posição $S(N-1,M-1)$, no caso de um terreno de $N \times M$ posições e em que a posição inicial é $(0,0)$. Para tal, S terá ser calculada da esquerda para a direita, e de cima para baixo no caso de se optar processar as linhas, ou então de cima para baixo e da esquerda para a direita no caso de se optar processar as colunas.

- a) [2,5]. Apresente um algoritmo que resolva este problema, recorrendo a programação dinâmica. Use linguagem natural estruturada, ou Java, para apresentar o algoritmo.
- b) [1,5]. Indique e justifique a complexidade espacial e temporal do algoritmo.

IMPORTANTE!

1. O enunciado deve obrigatoriamente ser entregue com as folhas de resposta, e identificado com o nome e número do aluno;
2. Responda às questões em folhas duplas, distribuindo-as da seguinte forma:

Questões 1 e 2

Questões 3 e 4

Questão 5