

Premissas

- ◆ As cinco principais características que suportam essa solução:
 1. Um conjunto de candidatos, de onde a solução é criada
 2. Uma função de seleção, 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

Implementação iterativa (Java)

```
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

Escalonamento de atividades: prova de optimalidade do algoritmo

- ◆ No algoritmo e exemplo dados, sejam:
 - A - conjunto inicial de atividades
 - a - atividade selecionada com fim mais cedo (a_1)
 - I - conj. de atividades incompatíveis com a ($\{a_3, a_4, a_7\}$)
 - C - conj. de atividades restantes ($\{a_2, a_5, a_6, a_8\}$)
- ◆ Do conjunto $\{a\} \cup I$, só pode ser selecionada no máximo uma atividade, pois são mutuamente incompatíveis *
 - * C/outra critério de ordenação (p.e. início mais cedo), podia não ser assim!
- ◆ Desse conjunto, escolhemos uma, que é o máximo possível
- ◆ A atividade escolhida (a) não tem incompatibilidade com as restantes (C), logo a escolha de a permite maximizar o nº de atividades que se podem escolher de C

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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 seleção
 - SE o conjunto solução se tornar 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 considerados

Prova de optimalidade

- ◆ Definição: Um sistema de moedas diz-se *canónico*, se o algoritmo ganancioso encontra sempre uma solução ótima para o problema do troco (com stock ilimitado).^[1]
- ◆ A maioria dos sistemas de moedas são canónicos (USA, EU, etc.).
- ◆ Teorema: Sendo $C = \{c_1, c_2, \dots, c_n\}$ as denominações do sistema de moedas, se o sistema for não canónico, o menor contra-exemplo situa-se na gama $c_3 + 1 < x < c_{n-1} + c_n$.^[1]
 - Logo basta fazer pesquisa exaustiva nesta gama para determinar se é canónico.
- ◆ Exemplo: Seja o sistema de moedas $C = \{1, 4, 5\}$.
 - Basta procurar contra-exemplos na gama $6 < x < 9$.
 - No caso $x = 7$, o algoritmo ganancioso dá a solução ótima $\{5, 1, 1\}$.
 - No caso $x = 8$, o alg. ganancioso dá $\{5, 1, 1, 1\}$ mas o ótimo é $\{4, 4\}$.
 - Logo o sistema não é canónico.

[1] Xuan Cai (2009). "Canonical Coin Systems for CHANGE-MAKING Problems". Proc. of the Ninth International Conference on Hybrid Intelligent Systems.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Escalonamento de atividades: verificação experimental

- ◆ O programa anexo “scheduling.cpp” gera instâncias aleatórias do problema (listas de atividades), aplica vários algoritmos de escalonamento gananciosos e compara com o resultado ótimo (obtido por algoritmo de pesquisa exaustiva)
- ◆ Se encontrar contra-exemplos, prova que o algoritmo em causa não garante o ótimo (sem termos de pensar muito na análise teórica ...)
- ◆ De facto, só os algoritmos gananciosos por fim mais cedo (cf. prova) e início mais tarde (simétrico do anterior) garantem o ótimo!

counter-example found for shortest interval: [[10, 14], [12, 18], [5, 12]]; expected result size=2; actual result size=1
counter-example found for earliest start: [[12, 25], [16, 16], [22, 28]]; expected result size=2; actual result size=1
counter-example found for latest finish: [[10, 27], [16, 25], [2, 14]]; expected result size=2; actual result size=1
counter-example found for fewest conflicts: [[5, 12], [89, 127], [25, 32], [80, 117], [31, 45], [48, 67], [44, 52], [68, 125], [54, 81], [47, 79], [44, 70], [27, 83], [27, 91], [11, 81], [5, 107]]; expected result size=5; actual result size=4
no counter-example found for earliest finish with up to 20 activities (1000 samples for each number of activities)
no counter-example found for latest start with up to 20 activities (1000 samples for each number of activities)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

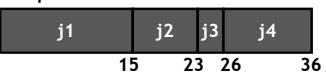
Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

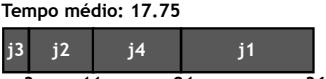
Escalonamento de atividades: minimizar tempo médio de conclusão

Variação do problema de escalonamento de atividades:

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

Tarefa	Tempo	
j1	15	
j2	8	
j3	3	
j4	10	

Tempo médio: 25

X

Tempo médio: 17.75

✓

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

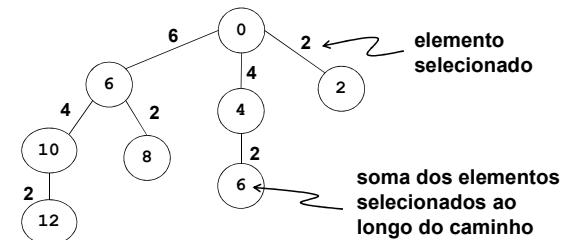
Nota sobre notação assintótica

- $T(n)$ - tempo de execução do algoritmo em função do tamanho n da entrada (no caso pior/melhor/médio)
- $T(n) = O(f(n))$ - $f(n)$ é um limite superior (*upper bound*) assintótico para $T(n)$, isto é,
 $\exists c > 0, n_0 > 0 \bullet \forall n > n_0 \bullet 0 \leq T(n) \leq c f(n)$
- $T(n) = \Theta(f(n))$ - $f(n)$ é um limite apertado (*tight bound*) assintótico para $T(n)$, isto é,
 $\exists c_1 > 0, c_2 > 0, n_0 > 0 \bullet \forall n > n_0 \bullet c_1 f(n) \leq T(n) \leq c_2 f(n)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Outras árvores de espaço de estados

- Outra possibilidade no problema da soma de subconjuntos: árvore n -ária em que, em cada nível, se escolhe o próximo valor a incluir.
- Para evitar repetir soluções, valores são escolhidas por ordem crescente ou decrescente.
- Cada nó representa uma solução candidata (total 2^n).



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Prova de optimalidade

- Tarefas: j_1, j_2, \dots, j_n , ordenadas por ordem de execução
- Durações: d_1, d_2, \dots, d_n
- Instantes de conclusão (fim): $f_1 = d_1, f_2 = d_1 + d_2, \dots$
- Tempo médio de conclusão das tarefas (custo):

$$\frac{\sum_{i=1}^n f_i}{n} = \frac{\sum_{i=1}^n (n-i+1)d_i}{n} = \frac{(n+1)\sum_{i=1}^n d_i - \sum_{i=1}^n i d_i}{n}$$
- Se existe $x > y$ tal que $d_x < d_y$, troca de j_x e j_y diminui custo da solução
- Assim, custo é minimizado se tarefas forem ordenadas tal que $d_1 \leq d_2 \leq \dots \leq d_n$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Nota sobre cálculo de $T(n)$ em funções recursivas (1/2)

- Merge (A, p, q, r)**
 - É óbvio que gasta um tempo $\Theta(n)$, com $n = r-p+1$ (tamanho da sequência a processar e nº de iter. do ciclo `for`).
- Merge-Sort (A, p, r)**
 - Vamos assumir que cada instrução tem um tempo de execução constante nas várias execuções
 - Para simplificar, vamos assumir que o tamanho da sequência original é uma potência de 2 (> 0)
 - $$T(n) = \begin{cases} c_1, & \text{if } n = 1 \\ c_2 + 2T\left(\frac{n}{2}\right) + c_3 n, & \text{if } n > 1 \end{cases} \quad (\text{com } n = r-p+1)$$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Outras otimizações

- Combinar com algoritmo ganancioso para procurar chegar mais rapidamente à solução (ou a uma boa solução quando se procura o ótimo)
 - Por exemplo, no problema do troco ou da soma de subconjuntos, começar a pesquisa pelos valores mais elevados
- Combinar com técnicas de memorização para evitar explorar repetidamente o mesmo nó, na presença de caminhos paralelos ou ciclos
 - Por exemplo, ao pesquisar num espaço de estados em forma de grafo, marcar os nós já visitados
- Em combinação com técnica de poda da pesquisa, podem melhorar o desempenho mas podem continuar a existir casos patológicos com tempo exponencial

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Memorização (memoization)

Para economizar tempo, basta aplicar a técnica de memorização (*memoization*), com *array* ou *hash map*.

```
long combMem(int n, int k) {
    // memory to store solutions (initially none)
    static long mem[100][100]; // n <= 99
    // if instance already solved, return from memory
    if (mem[n][k] != 0)
        return mem[n][k];
    // solve recursively
    long sol;
    if (k == 0 || k == n) sol = 1;
    else sol = combMem(n-1, k) + combMem(n-1, k-1);
    // memorize and return solution
    mem[n][k] = sol;
    return sol;
}
```

Implementação

```
long combDynProg(int n, int k) {
    int maxj = n - k;
    long c[1 + maxj];
    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];
}
```

Tempo: $T(n,k) = O(k(n-k))$
Espaço: $S(n,k) = O(n-k)$
($0 < k < n$, senão $O(1)$)

3º) Derivar fórmulas de cálculo

i	0	1	2	3	4	5	6=n
Sequência	s_i	$(-\infty)$	6	3	8	4	7
Tamanho	L_i	TL_i	3	1	2	0	1
Índ. 1º elem.	L_i	PL_i	2	3	4	-	5

- $TL_i = \max \{1+TL_k \mid i < k \leq n \wedge s_k > s_i\}$ ($i=n, \dots, 0$) ($\max \emptyset = 0$)
- PL_i = valor de k escolhido para o máximo na expressão de TL_i , caso exista, senão “-” ($i=n, \dots, 0$)
- Comprimento final: TL_0
- Solução final: $s_{PL_0}, s_{PL_0}, \dots$ (parando em “-”)
- Neste caso: (s_2, s_4, s_5) , isto é, $(3, 4, 7)$
- Solução “standard” é muito semelhante, mas parte de exploração em sentido inverso (do último para o 1º elemento)!

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

nC_k - Programação dinâmica

Para economizar memória, passa-se a abordagem *bottom-up*.

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

Calculando da esquerda para a direita, basta memorizar uma coluna.

ou

Calculando de cima para baixo, basta memorizar uma linha (diagonal).

Apêndice: Exemplos de Derivação de algoritmos

1. Explorar soluções p/ um exemplo
2. Tirar ilações e derivar estratégia
3. Derivar fórmulas de cálculo
4. Derivar algoritmo ou programa

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

4º) Derivar algoritmo ou programa

```
template <typename T>
void LIS(T s[], int n)
{
    int TL[n + 1] = {0}, PL[n + 1] = {0} /*undef*/;

    for (int i = n; i >= 0; i--)
        for (int k = i + 1; k <= n; k++)
            if (s[k - 1] > s[i - 1] && 1 + TL[k] > TL[i])
            {
                TL[i] = 1 + TL[k];
                PL[i] = k;
            }

    for (int i = PL[0]; i > 0; i = PL[i])
        cout << s[i - 1] << endl;
}
```

$T(n)=O(n^2)$
 $S(n)=O(n)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

3º) Derivar fórmulas de cálculo

	$i=0$	$v_0=-$	$C_{0,k}$	$P_{0,k}$	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$	$k=8=m$
$i=1$	$v_1=1$		$C_{1,k}$	$P_{1,k}$	0	1	2	3	4	5	6	7	8
$i=2$	$v_2=4$		$C_{2,k}$	$P_{2,k}$	-	1	1	1	1	1	1	1	1
$i=3$	$v_3=5$		$C_{3,k}$	$P_{3,k}$	0	1	2	3	1	1	2	3	2
					-	1	1	1	2	2	2	2	2

- > $C_{i,0} = 0$; $C_{0,k} = \infty$ (se $k > 0$); $P_{0,k} = P_{i,0}$ = indefinido (ou 0)
- > $C_{i,k} = C_{i-1,k}$, e $P_{i,k} = P_{i-1,k}$ para $i = 1, \dots, n$; $k = 1, \dots, v_i - 1$
- > $C_{i,k} = \min(C_{i-1,k}, 1 + C_{i,k-v_i})$ para $i = 1, \dots, n$; $k = v_i, \dots, m$
- > $P_{i,k} = P_{i-1,k}$ ou i , conforme se escolhe 1º ou 2º arg. de min
- > Cardinal final: $C_{n,m}$ Solução final: $v_{P_{n,m}}, v_{P_{n,m}}, v_{P_{n,m}}, \dots$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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 = i; j <= maxj; j++) → k(n-k) vezes
            c[j] += c[j-1];
    return c[maxj];
}
```

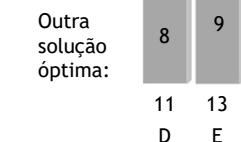
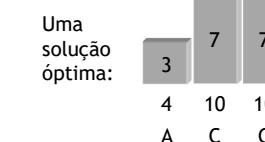
$T(n,k) = O(k(n-k))$
 $S(n,k) = O(n-k)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Exemplo

Itens	3	4	7	8	9
Tamanho	3	4	7	8	9
Valor	4	5	10	11	13
Nome	A	B	C	D	E

Capacidade da mochila: 17



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

4º) Derivar algoritmo ou programa

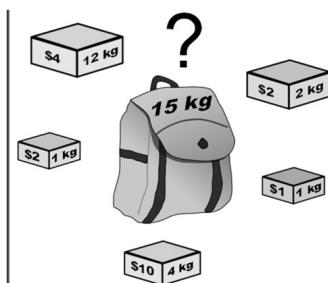
```
void troco(int m, int v[], int n)
{
    int C[1 + m] = {0}, P[1 + m] = {0} /*undef*/
    for (int k = 1; k <= m; k++)
        C[k] = m+1; /*mais alto que qq nº válido*/
    for (int i = 1; i <= n; i++)
        for (int k = v[i]-1; k <= m; k++)
            if (1 + C[k-v[i-1]] < C[k])
            {
                C[k] = 1 + C[k-v[i-1]];
                P[k] = i;
            }
    if (C[m] == m+1)
        cout << "Impossivel" << endl;
    else
        for (int k = m; k > 0; k = k-v[P[k]-1])
            cout << v[P[k]-1] << endl;
}
```

$T(n)=O(nm)$
 $S(n)=O(m)$

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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

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 ($N =$ nº de itens)
- ◆ Cálculo é eficiente em tempo e espaço se efectuado pela ordem apropriada

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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
- $\text{best}[M], \text{best}[M-\text{size}[\text{best}[M]]], \dots$ - itens seleccionados

11

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	-	-	$\text{cost}[k]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			$\text{best}[k]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	$\text{cost}[k]$	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			$\text{best}[k]$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	Iteração: i = 1; size[i] <= k <= M;																		
3	7	10	size[i] <= k AND val[i] + cost[k-i] > cost[k]? THEN best[k] = i; cost[k] = val[i] + cost[k-i];																		
4	8	11																			
5	9	13																			

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

15

Codificação

```
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;
        }
    }
}

// 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
 $\text{cost}[k-\text{size}[i]]$ já tem o valor da iteração i

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Formulação recursiva

12

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	-	-	$\text{cost}[k]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
			$\text{best}[k]$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	3	4	$\text{cost}[k]$	0	0	0	4	4	4	8	8	8	12	12	12	16	16	16	20	20	20
			$\text{best}[k]$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	4	5	$\text{cost}[k]$	0	0	0	4	5	5	8	9	10	12	13	14	16	17	18	20	21	22
			$\text{best}[k]$	0	0	0	1	2	2	1	2	2	1	2	2	1	2	2	1	2	2
3	7	10	$\text{cost}[k]$	0	0	0	4	5	5	8	10	10	12	14	15	16	18	20	20	22	24
			$\text{best}[k]$	0	0	0	1	2	2	1	3	2	1	3	3	1	3	3	1	3	3
4	8	11	$\text{cost}[k]$	0	0	0	4	5	5	8	10	11	12	14	15	16	18	20	21	22	24
			$\text{best}[k]$	0	0	0	1	2	2	1	3	4	1	3	3	1	3	3	4	3	3
5	9	13	$\text{cost}[k]$	0	0	0	4	5	5	8	10	11	13	14	15	17	18	20	21	23	24
			$\text{best}[k]$	0	0	0	1	2	2	1	3	4	5	3	3	5	3	3	4	5	3

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Números de Fibonacci

- $F = \{0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \dots\}$
- Fórmula de recorrência: $F(n) = F(n-1) + F(n-2)$, $n > 1$
 - $F(0) = 0$
 - $F(1) = 1$
- 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

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

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: $\max(l_i)$

19

Especificações

- ◆ Para provar que um algoritmo resolve corretamente um problema, precisamos de:
 - Especificação rigorosa do problema
 - Descrição rigorosa do algoritmo
- ◆ Muitos problemas podem ser especificados por um par:
 - Entradas: Dados de entrada e restrições associadas (pré-condições)
 - Saídas: Dados de saída e restrições associadas (pós-condições)
 - * Objetivos de maximização/minimização são redutíveis a restrições.

5

Pré-condições e pós-condições

- ◆ double squareRoot(double x)
- ◆ Pré-condições?
 - $x \geq 0$ (senão, implementação lança exceção)
- ◆ Pós-condições?
 - $RESULT * RESULT == x$... a menos de um certo erro!
 - $RESULT \geq 0$
- ◆ Algoritmo?
 - Método babilónico (derivado de Newton-Raphson), ...

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Cálculo para o exemplo dado

20

Correção parcial e total

- ◆ Correção parcial: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então, se terminar, produz saídas corretas, i.e., que obedecem às pós-condições.
- ◆ Correção total: se o algoritmo (ou programa) for executado com entradas que obedecem às pré-condições, então termina produzindo saídas que obedecem às pós-condições.

Resposta: (2, 3, 6)

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

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Pré-condições e pós-condições

- ◆ template <typename T>
int binarySearch(T a[], unsigned n, T x)
- ◆ Pré-condições?
 - Array a ordenado
 - $a \neq \text{NULL}$
 - Operadores de comparação definidos para o tipo T
- ◆ Pós-condições?
 - $(0 \leq \text{RESULT} < n \wedge a[\text{RESULT}] == x) \vee$
 - $(\text{RESULT} == -1 \wedge x \text{ não existe em } a)$

8

Pré-condições e pós-condições

- ◆ template <typename T>
void sort(T a[], unsigned n)
- ◆ Pré-condições?
 - a != NULL
 - Operadores de comparação definidos para o tipo T
- ◆ Pós-condições?
 - Array "a" está ordenado, isto é $a[0] \leq a[1] \leq \dots \leq a[n-1]$
 - Array final tem os mesmos elementos que o array inicial

9

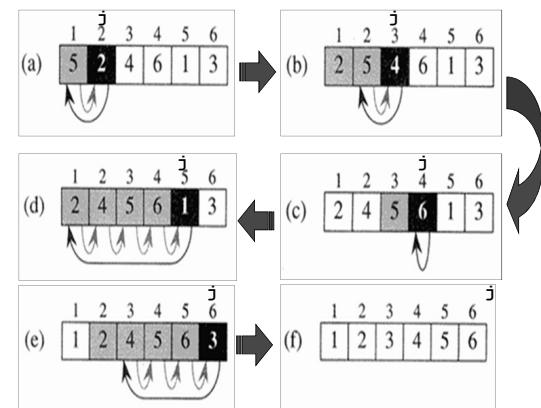
Invariante e variantes de ciclos

- ◆ A maioria dos algoritmos são iterativos, com um ciclo principal.
- ◆ Para provar que um ciclo está correto, temos de encontrar um invariante do ciclo - uma expressão booleana (nas variáveis do ciclo) 'sempre verdadeira' ao longo do ciclo, e mostrar que
 - é verdadeira inicialmente, i.e., é implicada pela pré-condição;
 - é mantida em cada iteração, i.e., é verdadeira no fim de cada iteração, assumindo que é verdadeira no início da iteração;
 - quando o ciclo termina, garante (implica) a pós-condição.
- ◆ Para provar que um ciclo termina, temos de encontrar um variante do ciclo - uma função (nas variáveis do ciclo)
 - inteira; positiva; estritamente decrescente. (porquê?)

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

11

Funcionamento



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Pré-condições e pós-condições

- ◆ template <typename T>
T max(T a[], unsigned n)
- ◆ Pré-condições?
 - a != NULL
 - n > 0
 - Operadores de comparação definidos para o tipo T
- ◆ Pós-condições?
 - RESULT pertence a a
 - Nenhum elemento de a é maior que RESULT

10

Exemplo 1: Insertion Sort

Ideia principal ...

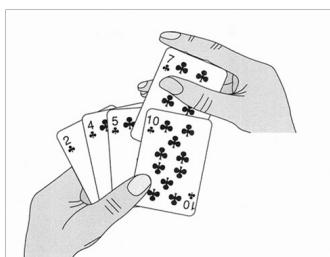


Figure 2.1 Sorting a hand of cards using insertion sort.

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

12

Algoritmo em pseudo-código

```
INSERTION-SORT(A, n)
for j = 2 to n
    key = A[j]
    // Insert A[j] into the sorted sequence A[1 .. j - 1].
    i = j - 1
    while i > 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

13

14

Análise (1/2)

- ◆ Invariante do ciclo principal [I(j)] ?

A[1, ..., j-1] contém os elementos originais, mas ordenados
(j = 2, ..., n+1)

- ✓ É válido inicialmente (j=2)
 - É óbvio que A[1..1] contém os elementos originais, mas ordenados
- ✓ É mantido em cada iteração
 - Assume-se que o invariante se verifica no início da iteração
 - O alg. insere A[j] na posição certa em A[1.. j] e incrementa j
 - Logo, no fim da iteração (com novo j), verifica-se o invariante
- ✓ No fim do ciclo (j = n+1), garante a pós-condição
 - Invariante refere-se a A[1.. n] ou seja todo o array
 - Logo, implica trivialmente a pós-condição, pois é coincidente

15

Exemplo 2: Binary Search

BINARY-SEARCH(A, n, x)

```

low ← 1
high ← n
while low ≤ high
    mid ← ⌊(low + high) / 2⌋
    if x = A[mid] then
        return mid
    else if x < A[mid] then
        high ← mid - 1
    else
        low ← mid + 1
return -1
  
```

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

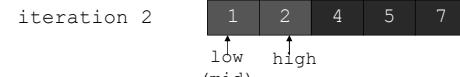
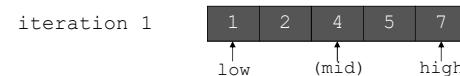
17

Exemplo de funcionamento

x: 3 A:

1	2	4	5	7
---	---	---	---	---

 n=5



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Análise (2/2)

- ◆ Variante do ciclo principal [v(j)] ?

n + 1 - j (j=2, ..., n+1)

- ✓ Inteiro, pois n e j são inteiros
- ✓ Não negativo, pois o valor máximo de j é n+1
- ✓ Estritamente decrescente, pois j é sempre incrementado
- ◆ Logo, o algoritmo está correto e termina (correção total)

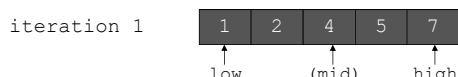
16

Exemplo de funcionamento

x: 5 A:

1	2	4	5	7
---	---	---	---	---

 n=5



Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

18

Análise (1/2)

- ◆ Invariante do ciclo [I (low, high)] ?

✗ só pode existir na área de pesquisa, entre low e high

- ✓ É válido inicialmente (low=1, high = n), pois a área de pesquisa é todo o array
- ✓ É mantido em cada iteração
 - ✓ Uma vez que se assume que o array está ordenado ...
 - ✓ quando se recua high, excluem-se apenas elementos > x
 - ✓ quando se avança low, excluem-se apenas elementos < x
- ✓ No fim do ciclo, garante a pós-condição
 - Se o ciclo é interrompido (A[mid] = x), garante-se a cláusula em que se encontra x
 - Se o ciclo for até ao fim, a área de pesquisa fica vazia, o que, pelo invariante, implica que x não existe em A

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

19

20

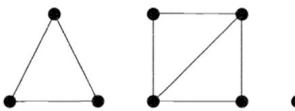
Análise (2/2)

- ◆ Variante do ciclo $[v(\text{low}, \text{high})]$?
 - Largura da área de pesquisa: $\boxed{\text{high} - \text{low} + 1}$
 - ✓ Inteiro, pois low e high são inteiros
 - ✓ Não negativo, pois no pior caso é $\text{low} = \text{high}$
 - ✓ Estritamente decrescente, pois em cada iteração ou aumenta-se low ou diminui-se high , estreitando-se a área de pesquisa
- Logo, o algoritmo está correto e termina (correção total)

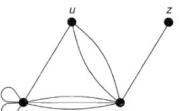
21

Grafo simples

- ◆ Grafo sem arestas paralelas (várias adjacências, para o mesmo par de vértices), nem anéis:



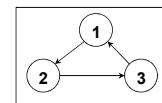
grafos simples



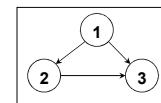
grafo complexo

Conectividade (2/2)

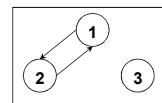
- ◆ Digrafo com a mesma propriedade: fortemente conexo, se p/ todo $v, w \in V$ existir em G um caminho de v para w , assim como de w para v
- ◆ Digrafo fracamente conexo: se o grafo não dirigido subjacente é conexo



Fortemente conexo



Fracamente conexo



Não conexo

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

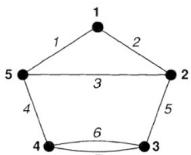
Revisão

- ◆ Quais são as 3 qualidades principais a analisar num algoritmo?
- ◆ Que técnicas de análise estática e dinâmica existem?
- ◆ O que são pré-condições e pós-condições? Para que servem?
- ◆ O que é correção parcial e total?
- ◆ O que são invariantes e variantes de ciclos? Para que servem?
- ◆ Quais são as 3 propriedades que temos de verificar num invariante de ciclo?
- ◆ Quais são as 3 propriedades que temos de verificar num variante de ciclo?

22

Grafo pesado

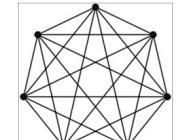
- ◆ As arestas são etiquetadas com um peso
 - Dependendo do tipo de grafo e problema, o peso pode representar uma distância, custo, etc.



Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

Densidade

- ◆ Grafo denso – $|E| \sim \Theta(V^2)$
 - Grafo completo – existe uma aresta entre qualquer par de vértices

Grafo completo com 7 vértices (K_7)

- ◆ Grafo esparsão – $|E| \sim \Theta(V)$



Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

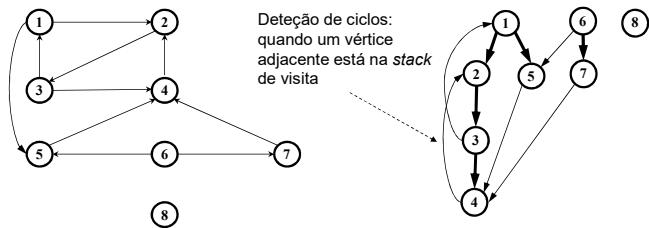
Técnicas de Concepção de Algoritmos, CAL - MIEIC/FEUP

13

10

Exemplo

Vértices numerados por (pré)ordem de visita e dispostos por profundidade de recursão:



Arestas a traço forte (que acederam a vértices ainda não visitados): floresta DFS constituída por uma ou mais árvores DFS (de expansão em profundidade).

Pseudo-código

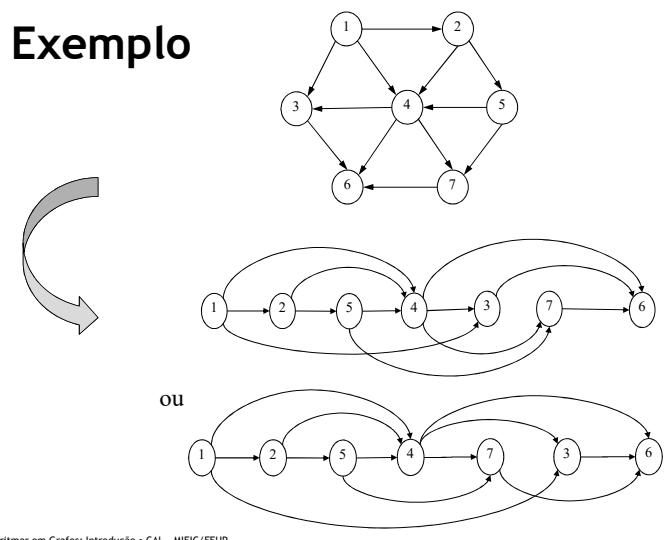
```
BFS(G, s) :
1.   for each v ∈ V do discovered(v) ← false
2.   Q ← ∅
3.   ENQUEUE(Q, s)
4.   discovered(s) ← true

5.   while Q ≠ ∅ do
6.       v ← DEQUEUE(Q)
7.       pre-process(v) ←
8.       for each w ∈ Adj(v) do
9.           if not discovered(w) then
10.              ENQUEUE(Q, w)
11.              discovered(w) ← true
12.       post-process(v) ←
```

Numa aplicação concreta de BFS, há processamento a fazer num destes pontos

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

Exemplo



Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

Pseudo-código

```
G = (V, E)
Adj(v) = {w | (v, w) ∈ E} (forall v ∈ V)

DFS(G) :
1.   for each v ∈ V
2.       visited(v) ← false
3.   for each v ∈ V
4.       if not visited(v)
5.           DFS-VISIT(G, v)

DFS-VISIT(G, v) :
1.   visited(v) ← true
2.   pre-process(v) ←
3.   for each w ∈ Adj(v)
4.       if not visited(w)
5.           DFS-VISIT(G, w)
6.   post-process(v) ←
```

Numa aplicação concreta de DFS, há processamento a fazer num destes pontos

12

Notas

- ◆ Para qualquer vértice v atingível a partir de s , o caminho na árvore BFS é o caminho mais curto no grafo (com menor número de arestas).
- ◆ BFS é um dos métodos mais simples e é o arquétipo para muitos algoritmos importantes de grafos.
 - Prim's Minimum-Spanning Tree
 - Dijkstra Single-Source Shortest-paths
- ◆ Se em vez de uma fila for usada uma pilha, obtém-se um algoritmo iterativo de visita em profundidade!

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

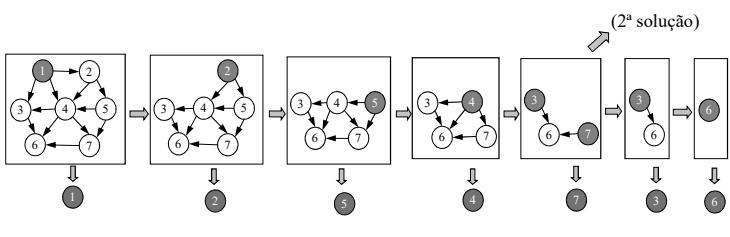
Método baseado em DFS

- ◆ Na visita em profundidade (DFS) de um DAG, a pós-ordem de visita dá uma ordenação topológica inversa
- ◆ São detetados ciclos quando vértice adjacente está na stack de visita
- ◆ Mas o método não é genérico, pois algumas ordenações topológicas válidas não podem ser obtidas desta forma
 - e.g., vértice 4 nunca fica entre vértices 3 e 2

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

Método geral

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



17

Algoritmo de ordenação topológica

TOP-SORT(in $G=(V,E)$, out T):

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

Algoritmos em Grafos: Introdução • CAL - MIEIC/FEUP

19

Algoritmos em Grafos: Caminho mais curto (Parte I)

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão

FEUP, MIEIC, CAL

FEUP

Universidade do Porto

Faculdade de Engenharia

CAL, Algoritmos em Grafos: Caminho mais curto

1

Algoritmo de ordenação topológica

- ◆ Refinamento do método geral:
 - simular eliminação atualizando 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)

18

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), o algoritmo pode ser executado em tempo $O(|V|+|E|)$
 - o corpo do ciclo de atualização 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

20

Índice

- Caminhos mais curtos de um vértice para todos os outros
 - Caso de grafos dirigidos não pesados
 - baseado em pesquisa em largura, $O(|V| + |E|)$
 - Caso de grafos dirigidos pesados
 - Dijkstra, algoritmo ganancioso, $O(|V| + |E|) \log |V|$
 - Caso de grafos dirigidos com arestas de peso negativo
 - Bellman-Ford, programação dinâmica, $O(|E| |V|)$
 - Caso de grafos dirigidos acíclicos
 - baseado em ordenação topológica, $O(|V| + |E|)$

FEUP Universidade do Porto
Faculdade de Engenharia

CAL, Algoritmos em Grafos: Caminho mais curto

2

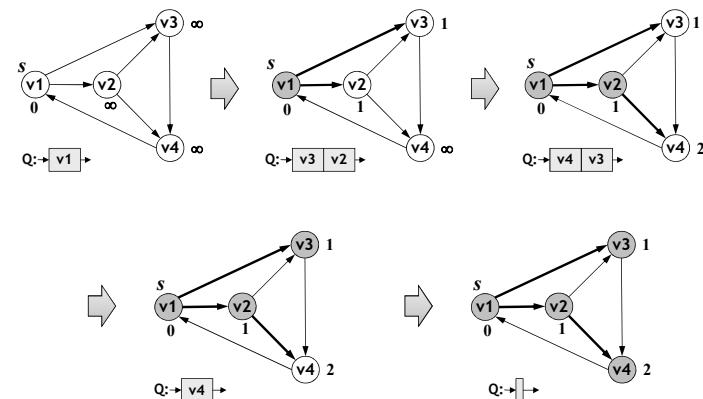
Aplicações

- Problemas de encaminhamento (*routing*)
 - Encontrar o melhor percurso numa rede viária
 - Nota: Algoritmo para encontrar caminho mais curto entre 2 pontos é baseado no algoritmo para encontrar caminhos mais curtos do ponto de partida para todos os outros
 - Encontrar o melhor percurso de avião
 - Encontrar o melhor percurso de metro
 - Encaminhamento de tráfego em redes informáticas
- Problemas de planeamento:
 - Planeamento de tarefas e respectivas dependências
 - Problemas operacionais, como minimização da cablagem necessária à ligação de pontos numa rede organizacional

Estruturas de dados

- Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar (ver bfs), 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 c/ nil)

Evolução da marcação do grafo



Caso de grafo dirigido não pesado

- Método básico (pesquisa em largura + cálculo de distâncias):
 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 o vértice v marcado com distância mínima
 3. Processar vértice v : analisar os adjacentes de v , marcando os que ainda não tinham sido alcançados ($\text{distância} \infty$) com distância de v mais 1
 4. Voltar 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, ...) é crucial para garantir eficiência
 - Distância fica definitiva / definida ao alcançar um vértice pela 1ª vez; ao alcançar por um 2º caminho, distância nunca diminui

Pseudo-código

```
SHORTEST-PATH-UNWEIGHTED (G= (V, E) , s) :
1.   for each v ∈ V do
2.     dist(v) ← ∞
3.     path(v) ← nil
4.   dist(s) ← 0
5.   Q ← ∅
6.   ENQUEUE (Q, s)
7.   while Q ≠ ∅ do
8.     v ← DEQUEUE (Q)
9.     for each w ∈ Adj (v) do
10.       if dist(w) = ∞ then
11.         ENQUEUE (Q, w)
12.         dist(w) ← dist(v) + 1
13.         path(w) ← v
```

Tempo de execução:
 $O(|E| + |V|)$

Espaço auxiliar:
 $O(|V|)$

Caso de grafo dirigido pesado ($\text{pesos} \geq 0$)

- Método básico semelhante ao caso de grafo não pesado
- Distância obtém-se somando pesos das arestas em vez de 1
- Próx. vértice a processar continua a ser o de distância mínima
 - Mas já não é necessariamente o mais antigo ⇒ Obriga a usar **fila de prioridades** (com mínimo à cabeça) em vez duma fila simples
 - Mas pode ser necessário rever em baixa a distância de um vértice alcançado e ainda não processado (vértice na fila) ⇒ Obriga a usar **fila de prioridades alteráveis**
 - Nota: A ordem é crucial para garantir que a distância ao vértice de partida dos vértices já processados não é mais alterada, assumindo que não há pesos negativos (ver análise adiante)
- É um algoritmo **ganancioso**: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância)

Algoritmo de Dijkstra (adaptado)

```

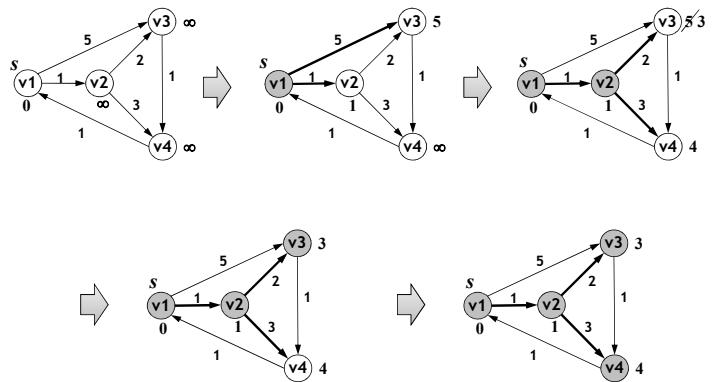
DIJKSTRA(G, s): // G=(V,E), s ∈ V
1. for each v ∈ V do
2.   dist(v) ← ∞
3.   path(v) ← nil
4. dist(s) ← 0
5. Q ← ∅ // min-priority queue
6. INSERT(Q, (s, 0)) // inserts s with key 0
7. while Q ≠ ∅ do
8.   v ← EXTRACT-MIN(Q) // greedy
9.   for each w ∈ Adj(v) do
10.     if dist(w) > dist(v) + weight(v,w) then
11.       dist(w) ← dist(v)+ weight(v,w)
12.       path(w) ← v
13.       if w ∉ Q then // old dist(w) was ∞
14.         INSERT(Q, (w, dist(w)))
15.     else
16.       DECREASE-KEY(Q, (w, dist(w)))
    
```

Tempo de execução:
 $O(|V| + |E|) * \log |V|$

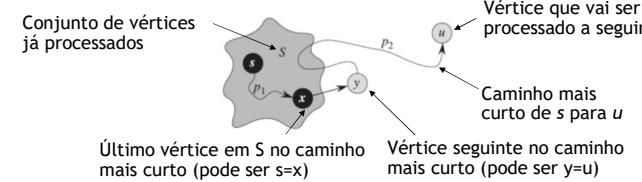
Análise do algoritmo de Dijkstra (1/2)

- Prova-se por indução o invariante do ciclo principal:
 - A distância conhecida dos vértices já processados ao vértice de partida é a distância mínima (logo não é mais alterada)
 - Mais precisamente, no momento em que se selecione um vértice u para processamento no início da ciclo, tem-se $d_{su} = \delta_{su}$
 - Notação: d - distância conhecida, δ - distância mínima
- Base (1º vértice) (inicialização do invariante)
 - O 1º vértice processado é o vértice de partida (s), com distância 0
 - Esta distância não pode ser reduzida (assumindo arestas de peso ≥ 0)
- Passo indutivo (k -ésimo vértice, $k > 1$) (manutenção do invariante)
 - Assume-se que o invariante se verifica para $k-1$
 - Ver slide seguinte

Evolução da marcação do grafo



Análise do algoritmo de Dijkstra (2/2)



- Uma vez que o algoritmo funciona por relaxamento sucessivo, $d_{su} \geq \delta_{su}$ (1)
- Sendo um caminho mais curto, e estando y antes de u , tem-se $\delta_{sy} \leq \delta_{su}$ (2)
- Sendo u o próximo vértice escolhido para processar, tem-se $d_{su} \leq d_{sy}$ (3)
- Uma vez que a aresta (x,y) foi analisada quando x foi processado com $d_{sx} = \delta_{sx}$ (pelo invariante), e (x,y) está num caminho mais curto, tem-se $d_{sy} = \delta_{sy}$ (4)
- Combinando (1), (2), (3) e (4), tem-se $\delta_{su} \leq d_{sy} \leq \delta_{sy} = \delta_{su}$ o que só é possível com $\delta_{su} = d_{su} = \delta_{sy} = \delta_{su}$, c.q.d.

Eficiência de DECREASE-KEY

- Suponhamos a fila de prioridades implementada com um *heap* (array) com o mínimo à cabeça e seja n o tamanho do *heap* (no máximo $|V|$)
- Método naïve: $O(n)$
 1. Procurar sequencialmente no array objeto cuja chave se quer alterar: $O(n)$
 2. Subir (ou descer) o objeto na árvore até restabelecer o invariante da árvore (cada nó menor ou igual que os filhos): $O(\log n)$
 - Total: $O(n) - Mau!$
- Método melhorado: $O(\log n)$
 - Cada objeto colocado no *heap* guarda a sua posição (índice) no *array*
 - Não é necessário o passo 1), logo o tempo total é $O(\log n)$
 - Introduz um *overhead* mínimo nas inserções e eliminações (quando se insere/move um objeto no *heap*, o seu índice tem de ser atualizado)
- Método otimizado: $O(1)$
 - Com Fibonacci Heaps consegue-se fazer DECREASE-KEY em tempo amortizado $O(1)$ (ver referências)

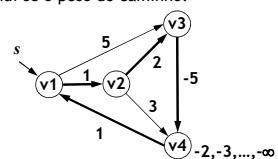
Eficiência do algoritmo de Dijkstra

- Tempo de execução é $O(|V| + |E| + |V|*\log|V| + |E|*\log|V|)$, ou simplesmente $O((|V|+|E|)*\log|V|)$
- $O(|V|*\log|V|)$ - extração e inserção na fila de prioridades
 - O $n^{\text{º}}$ de extrações e inserções é $|V|$
 - Cada operação destas pode ser feita em tempo logarítmico no tamanho da fila, que no máximo é $|V|$
- $O(|E|*\log|V|)$ - DECREASE-KEY
 - 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
- *Nota: O algoritmo proposto inicialmente por Dijkstra não mencionava filas de prioridades, e tinha uma eficiência $O(|V|^2)$

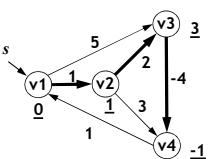
Caso de arestas com peso negativo

- Neste caso pode ser necessário processar cada vértice mais do que uma vez.
- Se existirem ciclos com peso negativo, o problema não tem solução.
- Não existindo ciclos com peso negativo, o problema é resolúvel em tempo $O(|V||E|)$ pelo **algoritmo de Bellman-Ford** (a seguir).

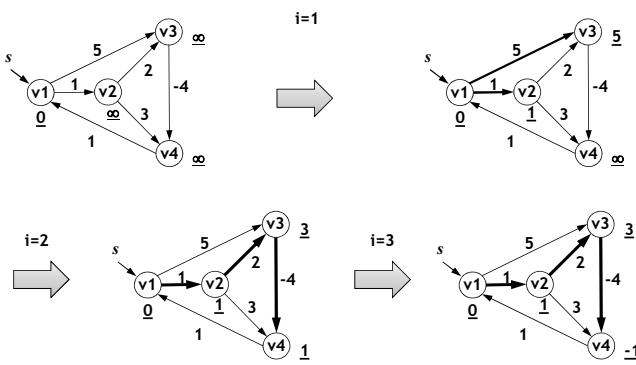
Sem solução, pois tem um ciclo de peso negativo (-1).
Percorrendo o ciclo várias vezes, diminui-se o peso do caminho.



Com solução, pois não tem ciclos de peso negativo.



Evolução da marcação do grafo



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 'novas' a entrar
 - Pode-se combinar a ordenação topológica com a atualização das distâncias e caminhos numa só passagem
 - Tempo de execução é o da ordenação topológica: $O(|V|+|E|)$
- Aplicações**
 - Processos irreversíveis
 - não se pode regressar a um estado passado (certas reações químicas)
 - deslocação entre dois pontos "em esqui" (sempre descendente)
 - Gestão de projetos
 - Projeto composto por atividades com precedências acíclicas (não se pode começar uma atividade sem ter acabado uma precedente)

Algoritmo de Bellman-Ford

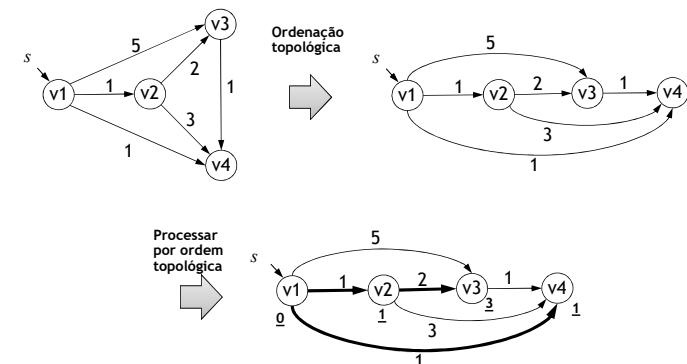
```
BELLMAN-FORD(G, s) : // G=(V,E), s ∈ V
1. for each v ∈ V do
2.   dist(v) ← ∞
3.   path(v) ← nil
4. dist(s) ← 0
5. for i = 1 to |V|-1 do
6.   for each (v, w) ∈ E do
7.     if dist(w) > dist(v) + weight(v,w) then
8.       dist(w) ← dist(v)+ weight(v,w)
9.       path(w) ← v
10.  for each (v, w) ∈ E do
11.    if dist(v) + weight(v,w) < dist(w) then
12.      fail("there are cycles of negative weight")
```

Tempo de execução:
 $O(|E| |V|)$

Análise do algoritmo de Bellman-Ford

- Em cada iteração i , o algoritmo processa todas as arestas e garante que encontra todos os caminhos mais curtos com até i arestas (e possivelmente alguns mais longos) (invariante do ciclo principal).
- Uma vez que o caminho mais comprido, sem ciclos, tem $|V|-1$ arestas, basta executar no máximo $|V|-1$ iterações do ciclo principal para assegurar que todos os caminhos mais curtos são encontrados.
- No final é executada mais uma iteração para ver se alguma distância pode ser melhorada; se for o caso, significa que há um caminho mais curto com $|V|$ arestas, o que só pode acontecer se existir pelo menos um ciclo de peso negativo.
- Podem ser efetuadas algumas melhorias ao algoritmo, mas que mantêm a complexidade temporal de $O(|V| |E|)$.
- É um caso de aplicação de programação dinâmica (Porquê?)

Exemplo

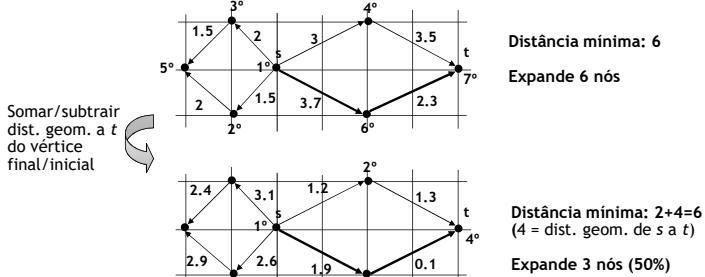


Pesquisa orientada (1/3)

- Algoritmo A*: escolher para processar o vértice v com valor mínimo de $d_{sv} + \pi_{vt}$, parando quando se vai processar o vértice t
 - d_{sv} - distância mínima conhecida de s a v (como no algoritmo de Dijkstra)
 - π_{vt} - estimativa por baixo da dist. mínima de v a t (função potencial)
- Em geral, não garante o ótimo
- Em certos casos, garante o ótimo, por exemplo:
 - Pesos das arestas são distâncias em km
 - π_{vt} é a distância Euclidiana (em linha reta) de v a t
 - Equivalente a aplicar o algoritmo de Dijkstra com pesos das arestas modificados $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt}$, somando-se no final π_{st} à distância mínima obtida de s para t (ver justificação a seguir).
 - Pode ser combinado com pesquisa bidirecional
 - Ganho (*speedup*) na prática é moderado

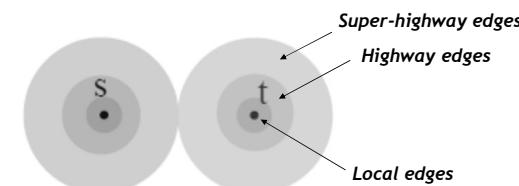
Pesquisa orientada (3/3)

Exemplo



Redes hierárquicas (highway networks) (2/2)

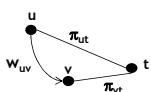
- Pesquisa é bidirecional e usa rede mais densa próximo de s e t e mais esparsa longe de s e t
- A pesquisa realiza-se em tempo da ordem de 1ms
- Exige pouco espaço adicional: um campo por aresta



Pesquisa orientada (2/3)

Justificação:

- Pela desigualdade triangular, garante-se $\pi_{ut} \leq w_{uv} + \pi_{vt}$, logo $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt} \geq 0$.
- O peso ao longo de um caminho $(s, v_1, v_2, \dots, v_k, t)$, fica igual ao do grafo original, acrescido de $\pi_{st} - \pi_{v_k t}$ (pois os potenciais intermédios cancelam-se)
- Logo, escolher o vértice v com menor $d_{sv} + \pi_{vt}$ no grafo modificado (A*), é o mesmo que escolher o vértice com menor d_{sv} no grafo original (Dijkstra)



Redes hierárquicas (highway networks) (1/2)

Pré-processamento decompõe a rede em vários níveis hierárquicos

- Analogia com mapa de estradas nacional e mapas de ruas locais
- Uma aresta (u, v) é classificada automaticamente como **highway edge** se existe pelo menos um par de nós s e t da rede tal que:
 - o caminho mais curto de s a t passa em (u, v) ;
 - u está a mais de H nós de distância de s ;
 - v está a mais de H nós de distância de t .
- H é um parâmetro configurável (por, exemplo, 40)
- Aplicável a mais níveis (local, highway, super-highway, etc.)
- Pré-processamento de mapa de USA ou Europa Ocidental pode ser efetuado em tempo da ordem de 15 minutos.

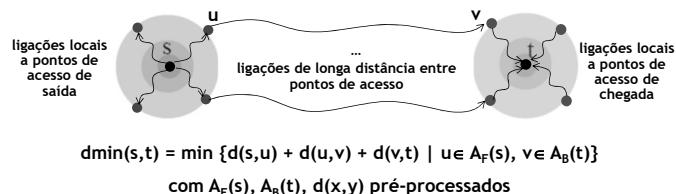
Nós de trânsito (transit-node routing)

- Pré-processamento determina:
 - nós de trânsito - nós tal que o caminho mais curto entre quaisquer 2 nós da rede que não estão "muito perto" entre si passa por pelo menos um dos nós de trânsito
 - Exemplo: acessos de auto-estradas
 - Há cerca de 10^4 nós de trânsito na Europa Ocidental e USA
 - Armazenam-se numa tabela as distâncias entre todos os pares de nós de trânsito
 - nós de acesso - para cada nó da rede, são os nós de trânsito mais próximos
 - Tipicamente há 10 nós de acesso por nó da rede
 - Armazenam-se numa tabela, para cada nó da rede, os nós de acesso e distâncias
 - Na verdade, determinam-se dois conjuntos de nós de acesso: nós de saída (*forward*, A_F) e nós de entrada (*backward*, A_B)

Nós de trânsito (*transit-node routing*)

- A pesquisa do caminho mais curto entre dois pontos afastados é reduzida a poucos *table lookups*, e realizada em tempo da ordem de $10\mu s$ (mas exige espaço de armazenamento adicional significativo)

- Obter nós de acesso dos pontos de partida (s) e chegada (t)
 - Para cada par (nó de acesso inicial (u), nó de acesso final (v)), obter distância de s a t em 3 *table lookups*

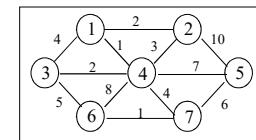


Caminho mais curto entre todos os pares de vértices

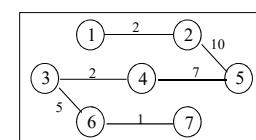
Caminho mais curto entre todos os pares de vértices

- Relevante por exemplo para pré-processamento de mapa de estradas
- Execução repetida do algoritmo de Dijkstra (ganancioso): $O(|V|(|V|+|E|) \log |V|)$
 - Bom se o grafo for esparsão ($|E| \sim |V|$), como é o caso das redes viárias
- Algoritmo de Floyd-Warshall, programação dinâmica: $\Theta(|V|^3)$
 - Melhor que o anterior se o grafo for denso ($|E| \sim |V|^2$)
 - Mesmo em grafos pouco densos pode ser melhor porque o código é mais simples
 - Baseia-se em matriz de adjacências $W[i,j]$ com pesos (∞ quando não há aresta; 0 quando $i = j$)
 - Calcula matriz de distâncias mínimas $D[i,j]$ e matriz $P[i,j]$ de predecessor no caminho mais curto de i para j

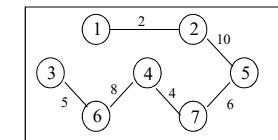
Exemplos de Árvores de expansão



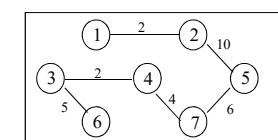
35



27



35



29

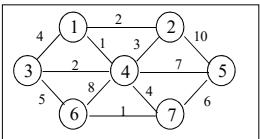
Algoritmo de Floyd-Warshall

- Invariante do ciclo principal: em cada iteração k (de 0 a $|V|$), $D[i,j]$ tem a distância mínima do vértice i a j , usando apenas vértices intermédios do conjunto $\{1, \dots, k\}$
- Inicialização ($k=0$):
 $D[i,j]^{(0)} = W[i,j]$ $P[i,j](0) = \text{nil}$
- Recorrência ($k=1, \dots, |V|$):
 $D[i,j]^{(k)} = \min(D[i,j]^{(k-1)}, D[i,k]^{(k-1)} + D[k,j]^{(k-1)})$
 - Valor de $P[i,j]^{(k)}$ é atualizado conforme o termo mínimo escolhido
- Para minimizar memória, pode-se atualizar a matriz em cada iteração k , em vez de criar uma nova

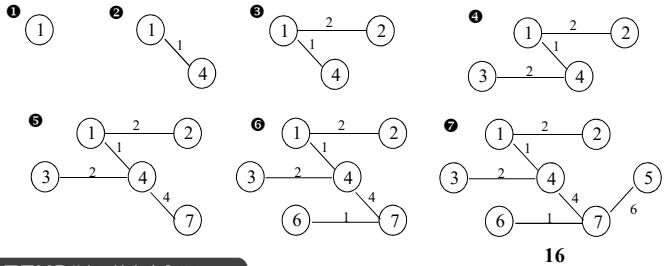
Algoritmo de Prim

- 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

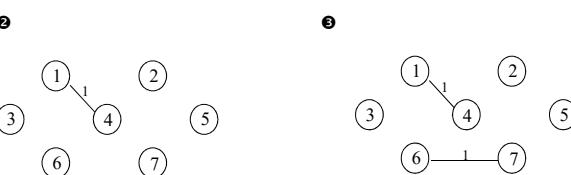
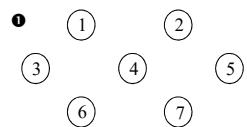
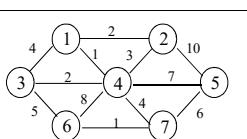
Evolução do algoritmo de Prim



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



Evolução do algoritmo de Kruskal



Pseudocódigo (Kruskal)

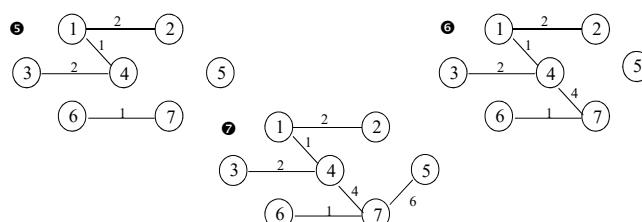
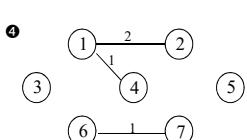
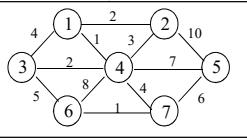
```
void kruskal() {
    int edgesAccepted = 0;
    PriorityQueue<Edge> h = readGraphIntoHeapArray();
    h.buildHeap();
    DisjSet<Vertex> s = new DisjSet(NUM_VERTICES);

    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);
        }
    }
}
```

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 árvores
 - 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
 - se são de conjuntos disjuntos, aceitar a aresta é aplicar-lhes uma União
- ❑ seleção 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

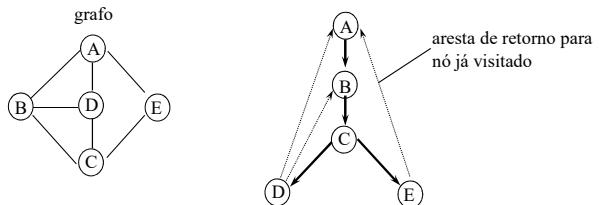


Referência e informação adicional

- “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006

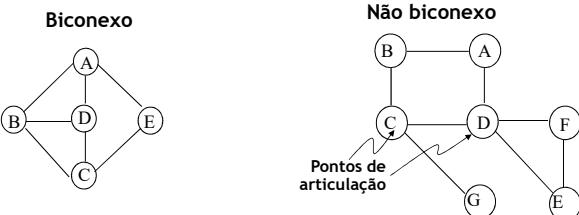
Conectividade

- Um grafo não dirigido é conexo se uma pesquisa em profundidade, a começar em um qualquer vértice, visita todos os vértices do grafo.



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



Cálculo de Low(v)

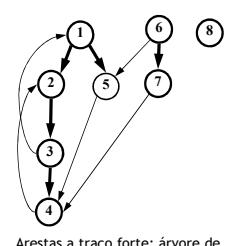
- Low(v) é mínimo de
 - Num(v)
 - o menor Num(w) de todas as arestas (v, w) de retorno
 - o menor Low(w) de todas as arestas (v, w) da árvore
- Na visita em profundidade, inicializa-se Low(v)=Num(v) antes de visitar adjacentes, e vai-se actualizando o valor de Low(v) a seguir a visita a cada adjacente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

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);
8:     // v passa a ser raiz duma árvore dfs
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):

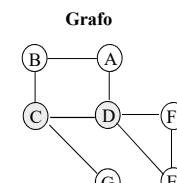


Na DFS podem ser produzidas várias árvores, porque a pesquisa pode ser repetida a partir de várias fontes (ao contrário da BFS que só produz uma). O conjunto das várias árvores é conhecido como Floresta DFS.

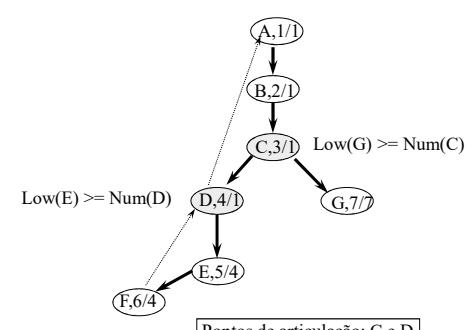
Algoritmo de detecção de pontos de articulação

- Início num vértice qualquer
- Pesquisa em profundidade, numerando os vértices ao visitá-los – Num(v), em pré-ordem (antes de visitar adjacentes)
- Para cada vértice v, na árvore de visita em profundidade, calcular 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 se tiver mais do que um filho na árvore

Exemplo



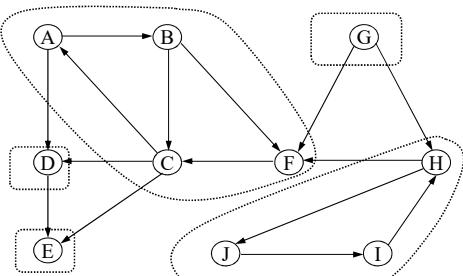
Uma árvore de expansão em profundidade
v, Num(v)/Low(v)



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);
            v.low = min(v.low, w.low);
            if(w.low >= v.num)
                System.out.println(v, "Ponto de articulação");
        }
        else
            if ( v.parent != w )      //aresta de retorno
                v.low = min(v.low, w.num);
    }
}
```

Componentes fortemente conexos



Componentes fortemente conexos

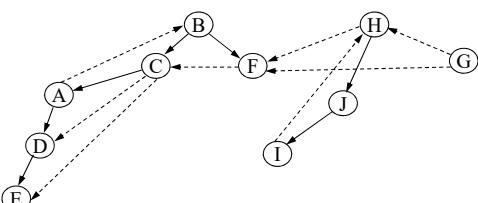
Método:

- 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 Gr)
- Segunda pesquisa em profundidade, em Gr, 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

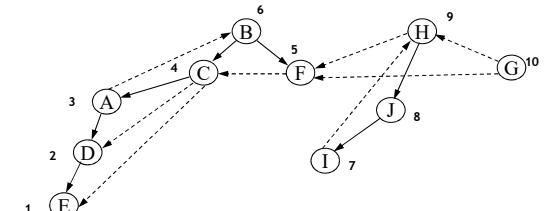
Grafos dirigidos

Árvore de expansão

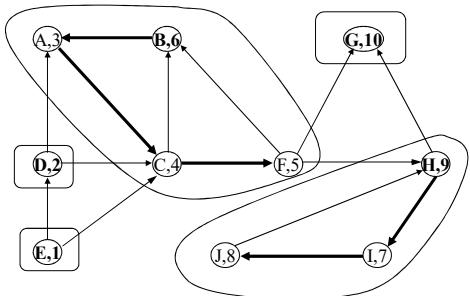
- Pesquisa em profundidade induz uma árvore/floresta de expansão
- Para além das arestas genuínas da árvore, há arestas para vértices já marcados
 - arestas de retorno para um antepassado – (A,B), (I,H)
 - arestas de avanço para um descendente – (C,D), (C,E)
 - arestas cruzadas para um nó não relacionado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



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

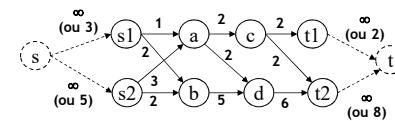
* Componentes fortemente conexos

■ 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 .

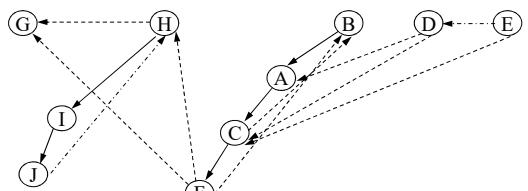
Redes com múltiplas fontes e poços

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



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

Componentes fortemente conexos



Travessia em pós-ordem de G_r

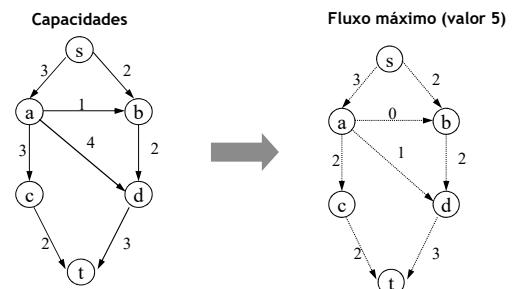
Componentes fortemente conexos:
 $\{G\}$, $\{H, I, J\}$, $\{B, A, C, F\}$, $\{D\}$, $\{E\}$

Referência e informação adicional

- “Data Structures and Algorithm Analysis in Java”, Second Edition, Mark Allen Weiss, Addison Wesley, 2006

Problema do fluxo máximo

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



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 :

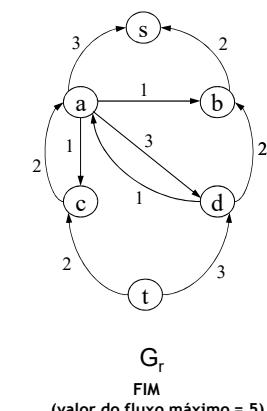
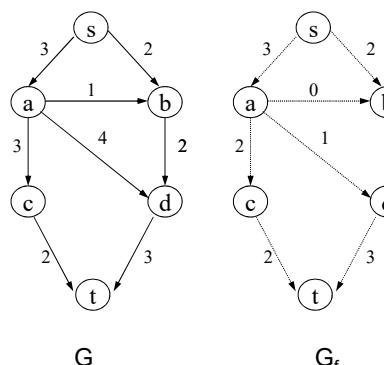
$$\max \sum_j f_{sj}$$

Algoritmo de Ford-Fulkerson (1955)

- Estruturas de dados:

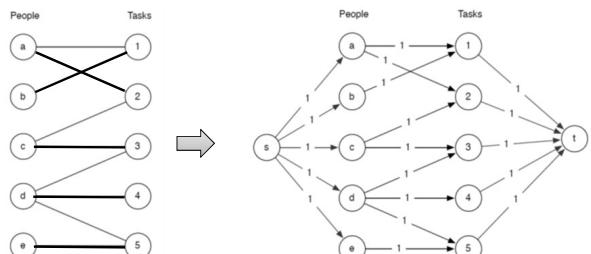
- G - grafo base de capacidades $c(v,w)$
- G_f - grafo de fluxos $f(v,w)$
 - inicialmente fluxos iguais a 0
 - no fim, tem o fluxo máximo
- G_r - grafo de resíduos (auxiliar)
 - para cada arco (v, w) em G com $c(v,w) > f(v,w)$, cria-se um arco no mesmo sentido em G_r de resíduo igual a $c(v,w) - f(v,w)$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

Exemplo: 2ª iteração



Exemplos de aplicação

- Rede de abastecimento de líquido ponto a ponto
- Tráfego entre dois pontos
- Emparelhamento máximo em grafos bipartidos (*maximum bipartite matching*)



Algoritmo de Ford-Fulkerson (1955)

- 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 correspondentes em G_f
 - Recalcular G_r

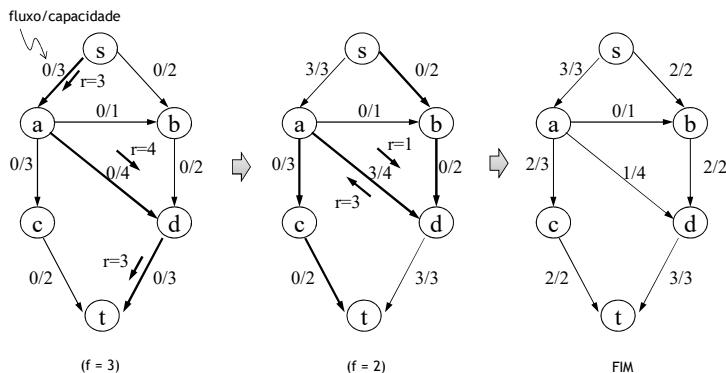
Análise do algoritmo de Ford-Fulkerson

- Se as capacidades forem números racionais, o algoritmo termina com o fluxo máximo
- Se as capacidades forem inteiros e o fluxo máximo F
 - Algoritmo tem a propriedade de integralidade: os fluxos finais são também inteiros
 - Bastam F iterações (fluxo aumenta pelo menos 1 por iteração)
 - Cada iteração pode ser feita em tempo $O(|E|)$
 - Tempo de execução total: $O(F |E|)$ - 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!
- Um caminho de aumento mais curto pode ser encontrado em tempo $O(|E|)$ através de pesquisa em largura
- Nº máximo de aumentos é $|E| \cdot |V|$ (ver referências)
- Tempo de execução: $O(|V| \cdot |E|^2)$

Exemplo



```
FindAugmentationPath(g, s, t): // Edmonds-Karp (breadth-first)
1. for v ∈ vertexSet(g) do visited(v) ← false
2. visited(s) ← true
3. Q ← ∅
4. ENQUEUE(Q, s)
5. while Q ≠ ∅ ∧ ¬ visited(t) do
6.   v ← DEQUEUE(Q)
7.   for e ∈ outgoing(v) do // direct residual edges
8.     TestAndVisit(Q, e, dest(e), capacity(e) - flow(e))
9.   for e ∈ incoming(v) do // reverse residual edges
10.    TestAndVisit(Q, e, orig(e), flow(e))
11. return visited(t)
```

```
TestAndVisit(Q, e, w, residual):
1. if ¬ visited(w) ∧ residual > 0 then
2.   visited(w) ← true
3.   path(w) ← e // previous edge in shortest path
4.   ENQUEUE(Q, w)
```

Implementação

- Para efetuar os cálculos num único grafo, guardam-se:
 - Em cada aresta:
 - orig: apontador para vértice de origem
 - dest: apontador para vértice de destino
 - capacity: capacidade da aresta
 - flow: fluxo na aresta
 - Em cada vértice:
 - outgoing: vetor de apontadores para arestas que saem do vértice
 - incoming: vetor de apontadores para arestas dirigidas ao vértice
 - visited: campo booleano usado na procura do caminho de aumento
 - path: apontador para aresta anterior no caminho de aumento
 - No grafo:
 - vertexSet: vetor de apontadores para vértices
- O grafo de resíduos é determinado “on the fly”
 - Arestas percorridas no sentido normal têm resíduo = capacidade - fluxo
 - Arestas percorridas no sentido inverso têm resíduo = fluxo

Pseudo-código

```
FordFulkerson(g, s, t):
1. ResetFlows(g)
2. tot ← 0
3. while FindAugmentationPath(g, s, t) do
4.   f ← FindMinResidualAlongPath(g, s, t)
5.   AugmentFlowAlongPath(g, s, t, f)
6.   tot ← tot + f
7. return tot
```

```
ResetFlows(g):
1. for v ∈ vertexSet(g) do
2.   flow(v) ← 0
```

```
FindMinResidualAlongPath(g, s, t):
1. f ← ∞
2. v ← t
3. while v ≠ s do
4.   e ← path(v)
5.   if dest(e) = v then // direct residual edge
6.     f ← min(f, capacity(e) - flow(e))
7.   v ← orig(e)
8.   else // reverse residual edge
9.     f ← min(f, flow(e))
10.  v ← dest(e)
11. return f
```

```
AugmentFlowAlongPath(g, s, t, f):
1. v ← t
2. while v ≠ s do
3.   e ← path(v)
4.   if dest(e) = v then // direct residual edge
5.     flow(e) ← flow(e) + f
6.   v ← orig(e)
7.   else // reverse residual edge
8.     flow(e) ← flow(e) - f
9.   v ← dest(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]	$O(n^2m^{1/2})$
1980	Gali-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]	$O(mn + n^{5/3} \log n)$
1992	King-Rao-Tarjan [38]	$O(mn + n^{2+e})$
1993	Phillips-Westbrook [42]	$O(mn \log_{m/n} n + n^2 (\log n)^{2+e})$
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)

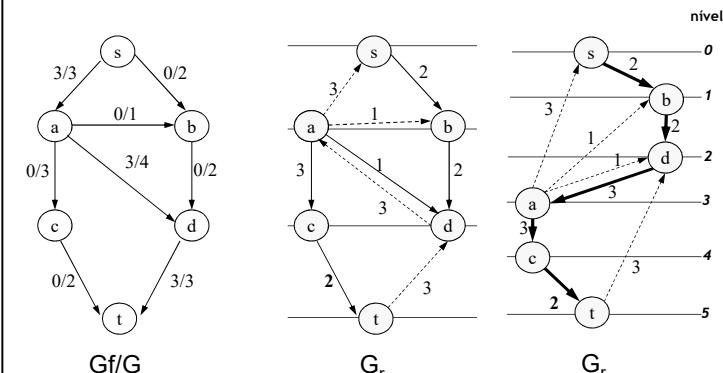
T. Asano and Y. Asano: recent developments in Maximum Flow Algorithms. Journal of the Operations Research, 1 (2000).

*Algoritmo de Dinic (1970)

- Refina algoritmo de Edmonds-Karp, evitando trabalho repetido a achar sucessivos caminhos de aumento de igual comprimento mínimo:

 - Iniciarizar os grafos de fluxos (G_f) e de resíduos (G_r) como antes
 - Calcular o nível de cada vértice, igual à distância mínima a s em G_r
 - Se nível(t) = ∞ , terminar
 - “Esconder” as arestas (u,v) de G_r em que nível(v) \neq 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!
 - 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
 - Se 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: 1ª iteração



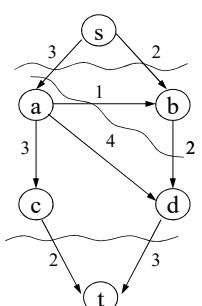
Não há mais caminhos de s para t deste nível (comprimento 3) \Rightarrow RENIVELAR

Dualidade entre fluxo máximo e corte mínimo

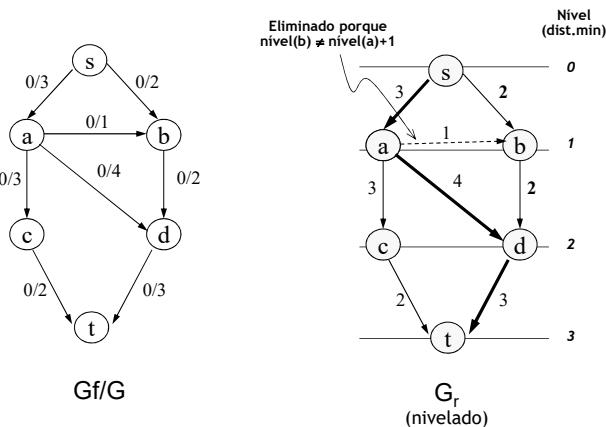
- Teorema: O valor do fluxo máximo numa rede de transporte é igual à capacidade do corte mínimo

- 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

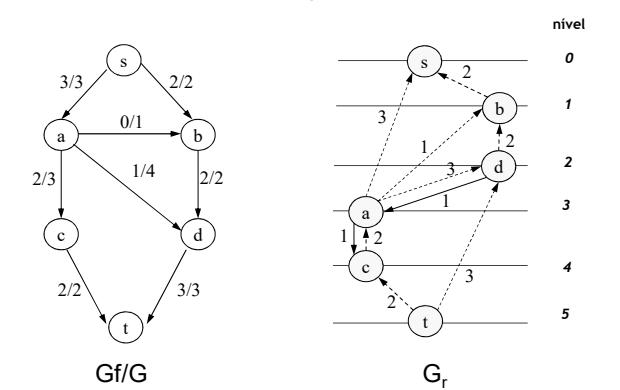
Cortes mínimos na rede do exemplo:



*Exemplo: estado inicial



*Exemplo: 2ª iteração

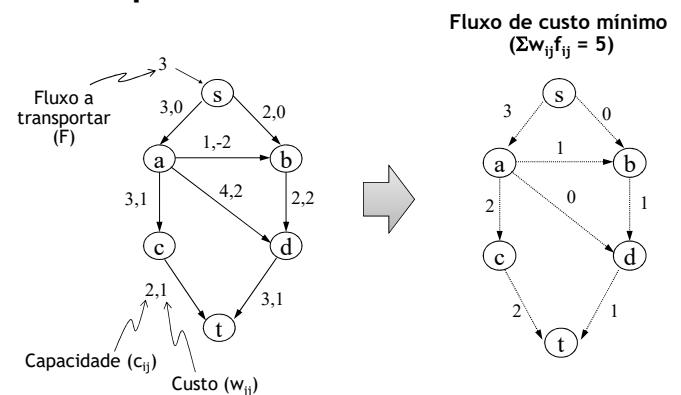


Não há mais caminhos de s para t neste nível e nível(t) = $|V|-1 \Rightarrow$ TERMINAR

* Eficiência do algoritmo de Dinic

- Passo 2 (cálculo do nível de cada vértice)
 - Cada execução pode ser feita em tempo $O(|E|)$ (assumindo $|E| > |V|$) por uma simples pesquisa em largura (ver explicação das referências)
- 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 (selecionar e aplicar um caminho de aumento)
 - O nº máximo de execuções (seleção 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, $O(|E| \cdot |V|) \sim O(|E|)$ para cada comprimento, multiplicado por $|V|$ comprimentos possíveis
 - Cada caminho de aumento pode ser encontrado em tempo $O(|V|)$ no grafo Gr (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)

Exemplo



*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 $|V|^{1/2}$ (vide referências)
- 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: $O(|V|^{1/2} |E|)$

Problema

- O objetivo é transportar uma certa quantidade F de fluxo (\leq máximo permitido pela rede) da fonte (s) para o poço (t), com um custo total mínimo
 - Para além da capacidade, arestas têm associado um custo (w_{ij} , 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 :

$$\begin{aligned} 0 \leq f_{ij} &\leq c_{ij}, \forall_{ij} \\ \sum_j f_{ij} &= \sum_j f_{ji}, \forall_{i \neq s,t} \\ \sum_j f_{sj} &= F \end{aligned}$$

Objetivo :

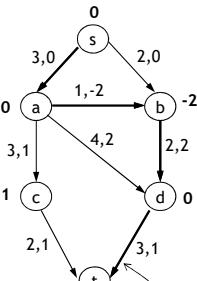
$$\min \sum_{ij} f_{ij} \times w_{ij}$$

Algoritmos

- Há muitos algoritmos propostos na literatura, aplicados à resolução deste problema, incluindo:
 - Cycle cancelling algorithms (*negative cycle optimality*)
 - Successive Shortest Path algorithms (*reduced cost optimality*)
 - Out-of-Kilter algorithms (*complimentary slackness*)
 - Network Simplex
 - Push/Relabel Algorithms
 - Dual Cancel and Tighten
 - Primal-Dual
 - ... entre outros!

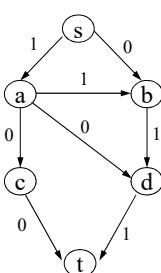
Exemplo (1/2)

Grafo inicial de resíduos e custos =
Grafo base de capacidades e custos



Caminho de custo mínimo de s a t
(fluxo = 1) (custo unitário = 1)

Grafo de fluxos resultante



Melhoramento

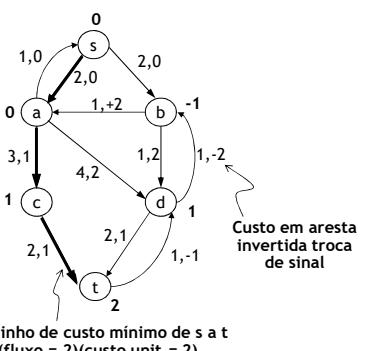
- Dificuldade na abordagem anterior: arestas de custo negativo no grafo de resíduos
 - Devido a custos iniciais negativos ou à inversão de arestas no grafo de resíduos
 - Obriga a usar algoritmo menos eficiente na procura do caminho de custo mínimo (Bellman-Ford $O(|V| |E|)$)
- Solução: converte-se o grafo de resíduos num equivalente (para efeito de encontrar caminho de custo mínimo) sem custos negativos
 - Na 1ª iteração usa-se algoritmo de Bellman-Ford $O(|V| |E|)$
 - Em todas as seguintes, usa-se algoritmo de Dijkstra $O(|E| \log |V|)$

Método dos caminhos de aumento mais curtos sucessivos

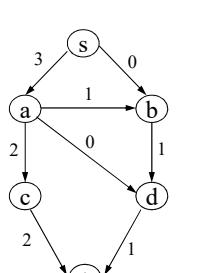
- Algoritmo ganancioso: no algoritmo de Ford-Fulkerson, escolhe-se em cada momento um caminho de aumento mais curto (no sentido de ter custo mínimo)
 - Pára-se quando se atinge o fluxo pretendido ou quando não há mais caminhos de aumento (neste caso dá um fluxo máximo de custo mínimo)
- Restrição: aplicável só a redes sem ciclos de custo negativo
 - Senão usa-se método mais genérico (cancelamento de ciclos negativos)
- Prova-se que dá a solução óptima (ver referências)

Exemplo (2/2)

Novo grafo de resíduos e custos

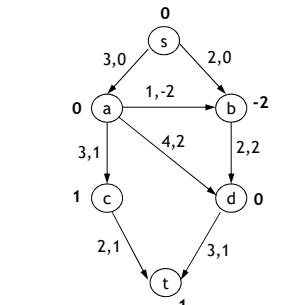


Grafo de fluxos resultante



Conversão do grafo de resíduos (1/2)

- No grafo de resíduos inicial, determinar a “distância” mínima de s a todos os vértices ($d(v)$)
 - Se existirem arestas (mas não ciclos) de peso negativo no grafo de resíduos inicial, usa-se o algoritmo de Bellman-Ford, de tempo $O(|E||V|)$
 - $d(v)$ também é chamado neste contexto o “potencial do nó v”

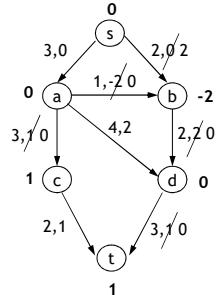


Conversão do grafo de resíduos (2/2)

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
 $d(v) \leq d(u) + w(u,v)$

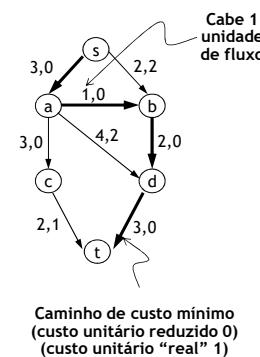
• O custo w' de um caminho de s a t , usando os custos reduzidos, é igual ao custo usando os custos antes da redução subtraído de $d(t)$ (demonstrar !)



Determinação do próximo caminho de aumento

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

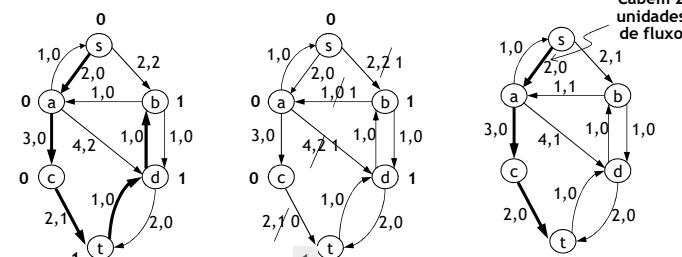
- Os caminhos de custo mínimo de s para t têm custo reduzido 0 e custo “real” (antes da redução) $d(t)$
- Como os caminhos de custo mínimo percorrem apenas arestas de custo 0, podem ser encontrados como uma pesquisa simples (DFS) em tempo linear
 - conceitualmente, eliminam-se arestas de custo > 0



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

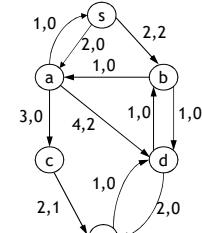
- Isto é, recalcular-se as distâncias mínimas (e.g. pelo algoritmo de Dijkstra!), e reduzem-se os custos (pode ser feito numa única passagem ...)



Aplicação do caminho de aumento

4. Aplicar o caminho de aumento

- Custo das arestas invertidas no grafo de resíduos é multiplicado por (-1)
- Só que $-1 \times 0 = 0$...
- Evita-se assim a introdução de arestas de custo negativo!

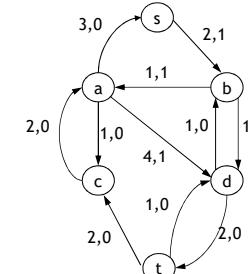


(quantidade atual do fluxo = 1)
(custo atual/real do fluxo = 1)

Repetição do processo

6. Aplicar o caminho de aumento de custo 0

- Actualiza-se o grafo de resíduos



Quantidade atual de fluxo = 3
= pretendido => FIM

Custo atual do fluxo = 5

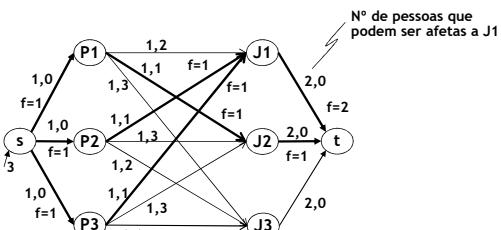
Eficiência temporal

- 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 inteiros, o nº máximo de iterações é F , pois em cada iteração o valor do fluxo é incrementado de uma unidade
- Tempo total fica $O(F |E| \log |V|)$

Aplicação a problemas de emparelhamento (2/2)

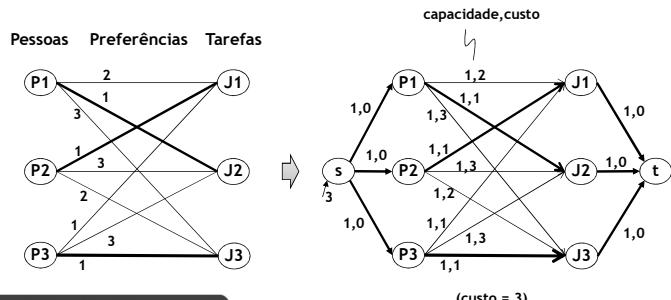
- E no caso de se poderem afetar várias pessoas à mesma tarefa?

- Por exemplo, no caso anterior, admitindo 2 pessoas por tarefa



Aplicação a problemas de emparelhamento (1/2)

- Problema de encontrar um emparelhamento de custo/peso mínimo num grafo bipartido (*minimum cost/weight bipartite matching*) (problema de afetação) pode ser reduzido ao problema de encontrar um fluxo de custo mínimo numa rede de transporte



Referências e informação adicional

- “Network Flows: Theory, Algorithms and Applications”, R. Ahuja, T. Magnanti & J. Orlin, Prentice-Hall, 1993
- “Efficient algorithms for shortest paths in sparse networks”. D. Johnson, J. ACM 24, 1 (Jan. 1977), 1-13

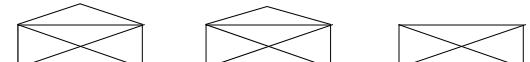
Algoritmos em Grafos: Circuitos de Euler e Problema do Carteiro Chinês

R. Rossetti, A. P. Rocha, L. Ferreira, J. P. Fernandes, F. Ramos, G. Leão

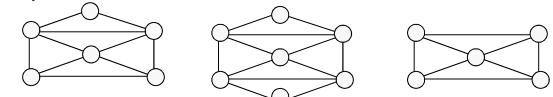
FEUP, MIEIC, CAL

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.



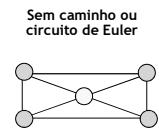
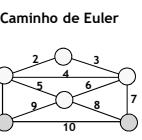
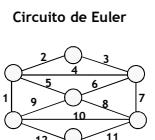
- Reformulação como problema em Teoria de Grafos: colocar um vértice em cada interseção



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

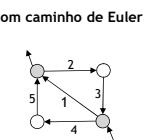
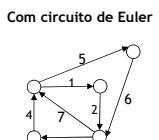
Condições necessárias e suficientes (1/2)

- Um grafo não dirigido contém um circuito de Euler se
 - é conexo e
 - cada vértice tem grau (nº de arestas incidentes) par.
- Um grafo não dirigido contém um caminho de Euler se
 - é conexo e
 - todos menos dois vértices têm grau par (estes dois vértices serão os vértices de início e fim do caminho).



Condições necessárias e suficientes (2/2)

- Um grafo dirigido contém um circuito de Euler se
 - é (fortemente) conexo e
 - cada vértice tem o mesmo grau de entrada e de saída.
- Um grafo dirigido contém um caminho de Euler se
 - é (fortemente) conexo e
 - 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 circuito de Euler

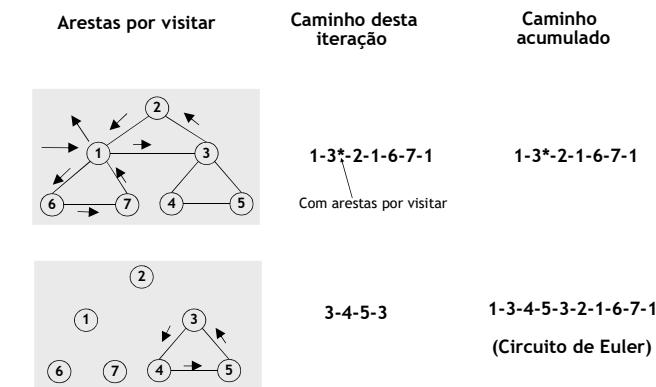
- Escolher um vértice qualquer e efetuar uma pesquisa em profundidade a partir desse vértice
 - Visitar vértice: se tiver arestas incidentes não visitadas, escolher uma dessas arestas, marcá-la como visitada, e visitar vértice adjacente
 - Se o grafo satisfizer as condições necessárias e suficientes, esta pesquisa termina necessariamente no vértice de partida, formando um circuito, embora não necessariamente de Euler
- Enquanto existirem arestas por visitar
 - Procurar o primeiro vértice no caminho (ciclo) obtido até ao momento que possua uma aresta não percorrida
 - Lançar uma sub-pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas)
 - Inserir o resultado (ciclo) no caminho principal

Estruturas de dados e eficiência temporal

- 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 efetuar inserções em tempo constante

Exemplo em grafo não dirigido

Exemplo em grafo não dirigido

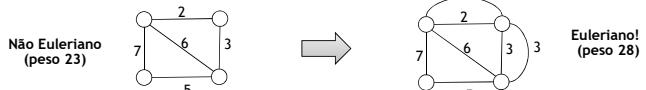


Problema do carteiro chinês (*Chinese postman problem*)

- 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 atravessasse cada aresta de G pelo menos uma vez.
 - A um caminho assim chama-se *percurso ótimo do carteiro Chinês*.
 - A um caminho fechado (não necessariamente de peso mínimo) que atravessasse cada aresta pelo menos uma vez chama-se *percurso do carteiro*.
- Problema estudado pela primeira vez por 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.
- Resolúvel em tempo polinomial para grafos dirigidos ou não dirigidos, mas infelizmente o problema é NP-completo (tempo exponencial) quando se combinam arestas dirigidas com arestas não dirigidas (grafos mistos)
 - Exemplo: percurso do camião do lixo, quando algumas ruas têm sentidos únicos

Abordagem

- Se o grafo G for Euleriano, a solução é trivial, pois qualquer circuito de Euler é um percurso ótimo do carteiro Chinês.
 - Cada aresta é percorrida exatamente uma vez.
- Se o grafo G não for Euleriano, pode-se construir um grafo Euleriano G^* duplicando algumas arestas de G , selecionadas por forma a conseguir um grafo Euleriano com peso total mínimo.



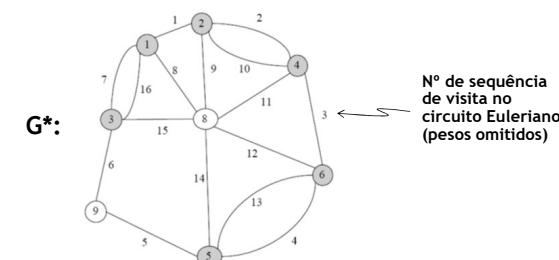
Método para grafos não dirigidos (2/4)

- Passo 2: Achar os caminhos mais curtos e distâncias mínimas entre todos os pares de vértices de grau ímpar em G .

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

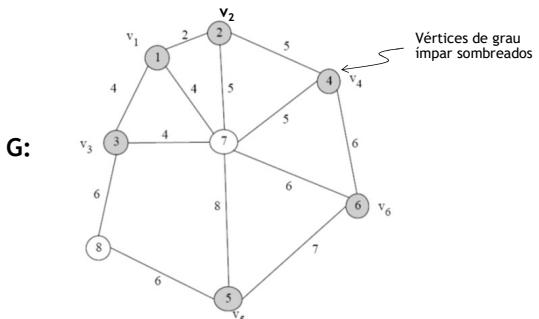
Método para grafos não dirigidos (4/4)

- Passo 5: Para cada par (u, v) no emparelhamento perfeito obtido, adicionar pseudo-arestas (arestas paralelas duplicadas) a G ao longo de um caminho mais curto entre u e v . Seja G^* o grafo resultante.
- Passo 6: Achar um circuito de Euler em G^* . Este circuito é um percurso ótimo do carteiro Chinês.



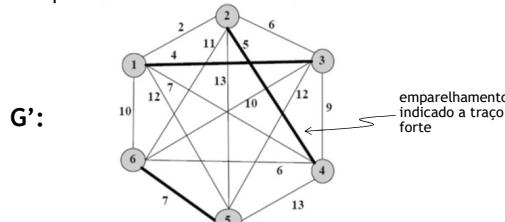
Método para grafos não dirigidos (1/4)

- Passo 1: Achar todos os vértices de grau ímpar em G . Seja k o nº (par!) destes vértices. Se $k=0$, fazer $G^*=G$ e saltar para o passo 6.



Método para grafos não dirigidos (3/4)

- Passo 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.
- Passo 4: Encontrar um emparelhamento perfeito (envolvendo todos os vértices) de peso mínimo em G' . Isto corresponde a emparelhar os vértices de grau ímpar de G , minimizando a soma das distâncias entre vértices emparelhados.

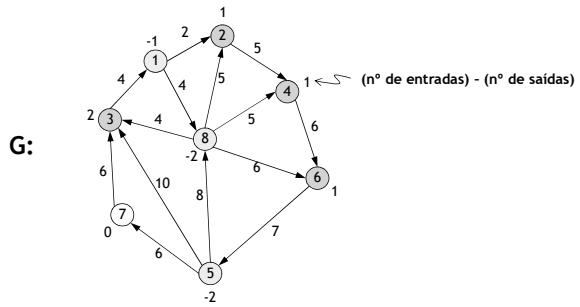


* Realização do passo 4 - Emparelhamento perfeito de peso mínimo

- 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).

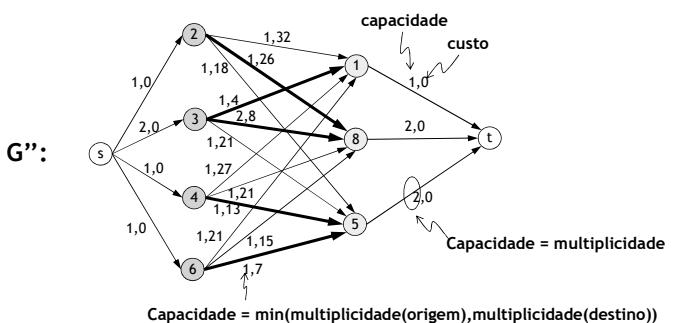
Método para grafos dirigidos (1/4)

- Passo 1: No grafo G dado, identificar os vértices com n°s diferentes de arestas a entrar e a sair



Método para grafos dirigidos (3/4)

- Passo 3: Formular problema de emparelhamento óptimo como problema de fluxo máximo de custo mínimo e resolver.



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

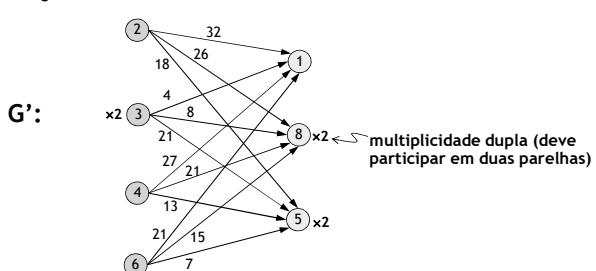
Casamentos estáveis

- 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

Método para grafos dirigidos (2/4)

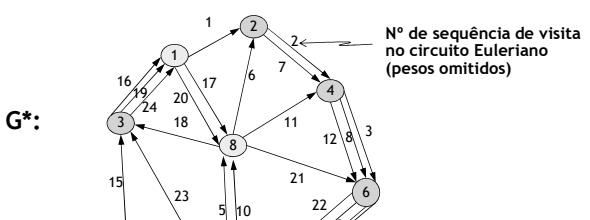
- Passo 2: Determinar os caminhos mais curtos de vértices que têm défice de saídas para vértices que têm défice de entradas e representar as distâncias respectivas num grafo bipartido G' .

Vértices são anotados com multiplicidade (nº de pares em que deve participar) igual ao défice absoluto



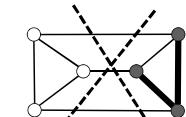
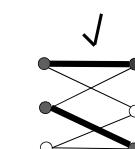
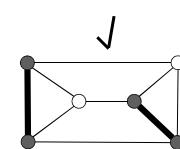
Método para grafos dirigidos (4/4)

- Passo 4: Obter grafo Euleriano G^* , duplicando em G os caminhos mais curtos entre os vértices emparelhados no passo 3, e obter um circuito Euleriano.



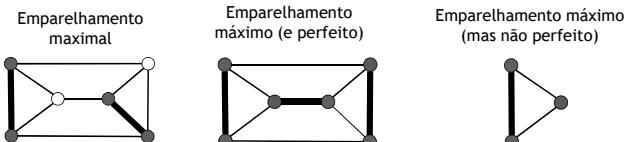
Conceito de emparelhamento

- 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
- Também chamado conjunto de arestas independentes



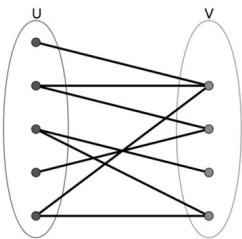
Características de emparelhamentos

- **Emparelhamento maximal:** não pode ser aumentado
- **Emparelhamento máximo:** tem tamanho máximo
 - Não é necessariamente único
 - É necessariamente maximal
 - O número $v(G)$ é o tamanho do emparelhamento máximo
- **Emparelhamento perfeito:** inclui todos os vértices

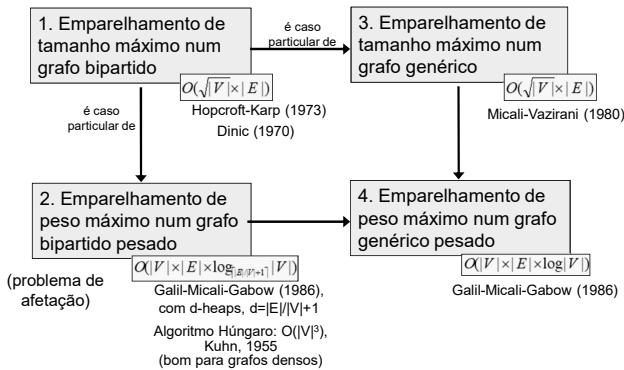


Conceito de grafo bipartido

- **Grafo bipartido** (ou bigrafo): grafo cujos vértices podem ser divididos em dois conjuntos disjuntos U e V , tal que todas as arestas de G ligam um vértice u de U a um vértice v de V .

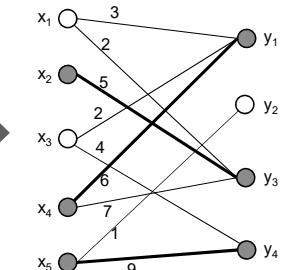
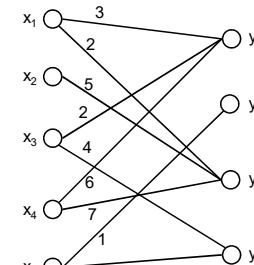


Problemas de emparelhamento



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

e.g. pessoas candidatam-se a empregos



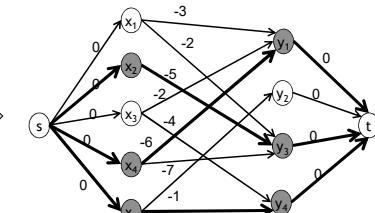
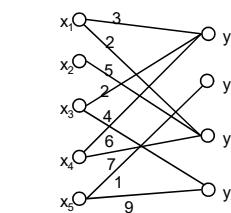
Peso total: 20
(estratégia gananciosa daria 19)

Redução a problemas em redes de transporte

- Problemas de emparelhamento em grafos bipartidos são reduzíveis 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 reduzíveis a grafos bipartidos
 - Basta fazer uma pesquisa em largura, a qual gera uma floresta de 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 *

Resolução como problema de fluxo máximo de custo mínimo

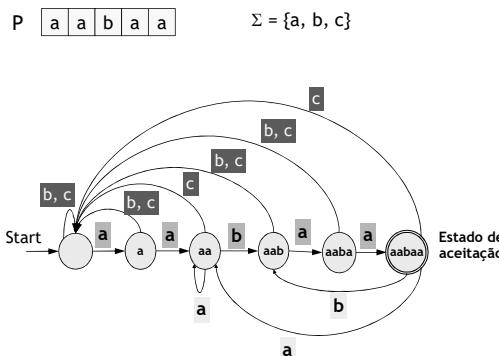
- Capacidades unitárias (logo só se mostram custos e não capacidades)
- Custo do transporte = simétrico do peso do emparelhamento (origina arestas de custo negativo, mas não há ciclos)
- Aplica-se método dos caminhos de aumento de custo mínimo; para-se quando o próximo caminho de aumento tem custo real ≥ 0



Algoritmos

- Algoritmo *naïve*
 - 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
 - Efetua 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 $O(|T| + |P|)$

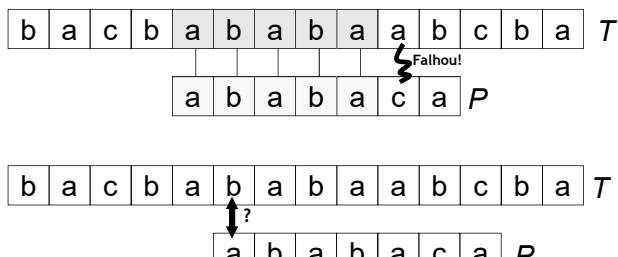
Autómato finito correspondente ao padrão



Pré-processamento do padrão

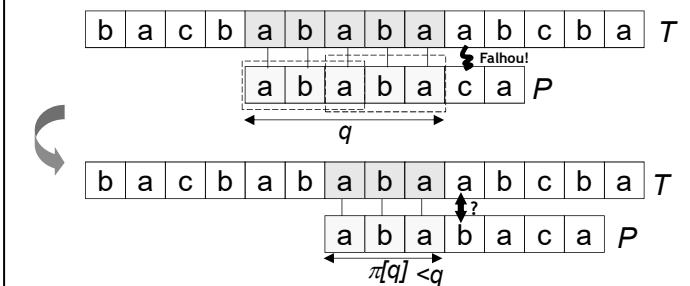
- Compara-se o padrão com deslocações do mesmo, para determinar a função prefixo
- $$\pi[q] = \max \{k: 0 \leq k < q \text{ e } P[1..k] = P[(q-k+1)..q]\}$$
- $q = 1, \dots, |P|$
 - $P[i..j]$ - substring entre índices i e j
 - Índices a começar em 1
 - $\pi[q]$ é o comprimento do maior prefixo de P que é um sufixo próprio do prefixo de P de comprimento q

Algoritmo naïve



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

Algoritmo de Knuth-Morris-Pratt

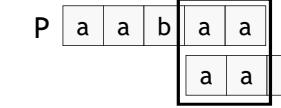
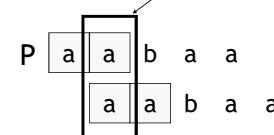


Desloca-se o padrão para a direita de uma forma que permite continuar a comparação na mesma posição do texto! Evita comparações inúteis!

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

Pré-processamento do padrão

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.
```

Formulação recursiva

- $D[i, j] = \text{EditDistance}(P[1..i], T[1..j]), 0 \leq i \leq |P|, 0 \leq j \leq |T|$
- Condições fronteira:
 - $D[0, j] = j, D[i, 0] = i$ (porquê?)
- 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 $T[j]$)

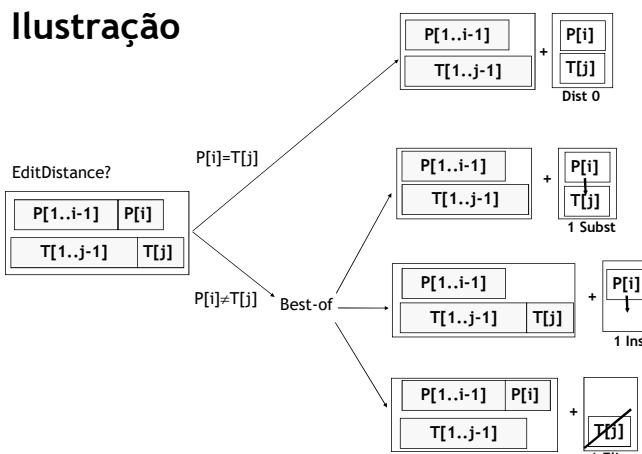
Matriz de programação dinâmica

		T												
		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
a	1	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	1	2	3	4	5	6	7	
h	8	7	6	5	4	3	2	1	2	3	4	5	6	
i	9	8	7	6	5	4	3	2	1	2	3	4	5	
j	10	9	8	7	6	5	4	3	2	1	2	3	4	
k	11	10	9	8	7	6	5	4	3	2	1	2	3	
l	12	11	10	9	8	7	6	5	4	3	2	1	3	

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  $\pi$ 
```

Ilustração



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, ← Tem D[i-1,j-1]
Ainda tem valor anterior D[i-1,j] → D[j],
D[j-1]) ← Já tem valor da iteração
                corrente, i.e., D[i, j-1]
            old = D[j]
            D[j] = new
    // finalização
    return D[|T|]
}
```

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 (sem perdas)
 - Keyword encoding
 - Run-length encoding (RLE)
 - Huffman codes
- Na prática aplica-se muitas vezes uma combinação de técnicas

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% do original)

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} (tamanho 2)
- Compressão de texto com códigos de Huffman
- Criptografia

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

Run-length encoding (RLE)

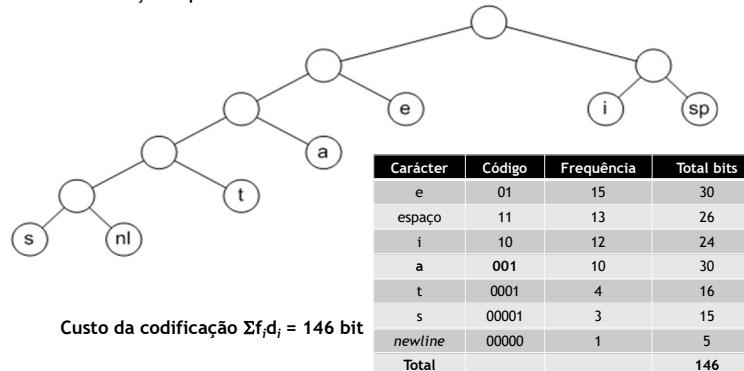
- Tipicamente utilizado quando o mesmo padrão/letra surge muitas vezes seguidas numa sequência de dados;
- Não é comum em texto, mas em muitos outros tipos de dados (por exemplo: imagem, vídeo)
- Técnica utilizada em muitas aplicações comuns. Basicamente, uma sequência de caracteres que se repetem é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

AAAAAAAAAA → *A10
AABBBBBBBBAMMMKKKKKKKM → AA*B8AMM*K9M

Algoritmo de Huffman

Codificação variável (cont)

- Codificação óptima...



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 (*algoritmo ganancioso*)
- Repetir o passo anterior até haver uma só árvore
- Empates são resolvidos aleatoriamente

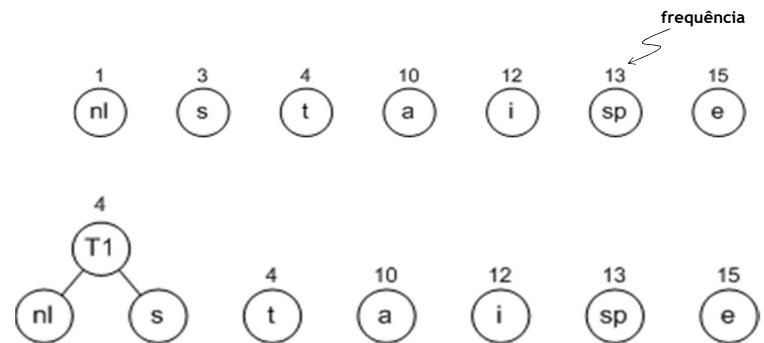
Codificação constante

- Código de tamanho fixo.
 - Se $|\text{alfabeto}| = C \rightarrow$ código com $\lceil \log_2(C) \rceil$ bits
 - Ex: caracteres ASCII visíveis $\cong 100 \rightarrow$ necessário código de 7 bits
- Representação possível
 - Árvore binária com caracteres só nas folhas
 - Na descodificação:
 - se é folha, então encontrou-se o carácter
 - se o bit correte do código for 0, visita-se a sub-árvore esquerda
 - se o bit correte do código for 1, visita-se a sub-árvore direita

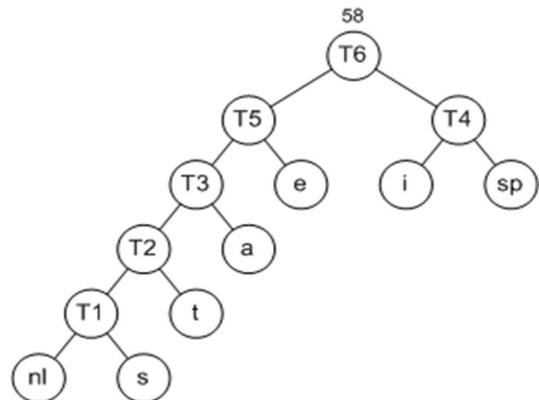
Códigos de Huffman

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

Algoritmo de Huffman (exemplo)



Algoritmo de Huffman (exemplo)



A classe de problemas P

- A classe de problemas P é constituída por todos os problemas de decisão que podem ser resolvidos em tempo polinomial
- A generalidade dos problemas em grafos que temos abordado pertencem a esta classe (na versão de problema de decisão)
 - Caminho mais curto
 - Árvore de expansão mínima
 - Fluxo máximo
 - Fluxo de custo mínimo
 - Circuito de Euler
 - Problema do carteiro chinês
 - Problemas de emparelhamento
 - ...

Problemas de Decisão

- Um problema de decisão é um problema cujo *output* ou resposta deve ser um simples “SIM” ou “NÃO” (ou derivativos do tipo “V/F”, “0/1”, “aceitar/rejeitar”, etc.)
- Muitos problemas práticos são problemas de optimização (maximizar ou minimizar alguma métrica), mas 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 G ?” pode ser expresso como “Dado um grafo G e um inteiro k , é possível colorir G com k cores?”

A classe de problemas NP

Algoritmo de Huffman

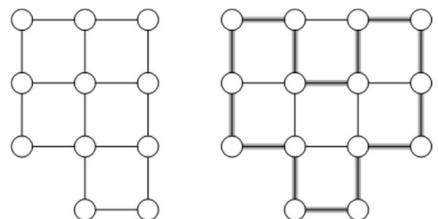
- Construção da árvore binária:

Algoritmo de Huffman:
Entrada: Um conjunto C de n caracteres.
Saída: Árvore de Huffman.

```
n ← |C|;  
Q ← C;  
para i ← 1 até n – 1 faça  
    CriaNo(z);  
    x ← z.esq ← ExtraiMinimo(Q);  
    y ← z.dir ← ExtraiMinimo(Q);  
    f[z] ← f[x] + f[y];  
    Insere(Q, z);  
    retorna ExtraiMinimo(Q);
```

Problema do Circuito Hamiltoniano

- Problema do circuito Hamiltoniano não dirigido (*Undirected Hamiltonian Cycle - UHC*): verificar se um grafo não dirigido dado G , é Hamiltoniano, isto é, tem um ciclo (ou circuito) que visita cada vértice exatamente uma vez.



As classes de problemas NP-completos e NP-difíceis

Classe de problemas NP

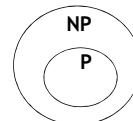
- A **classe de problemas NP** (*nondeterministic polynomial*) é definida por todos os problemas que podem ser verificados por um algoritmo de tempo polinomial
 - “Verificados” no sentido explicado no slide anterior
 - Não confundir **resolução** em tempo polinomial (P) com **verificação** em tempo polinomial (NP)!
 - * O termo “*nondeterministic*” provém da definição inicial da classe NP em termos de máquinas de Turing não deterministas, capazes de não deterministicamente conjecturar o valor do certificado (em geral conjecturar uma *string*) e verificar o depois, em tempo polinomial (em geral, verificar se a string faz parte da linguagem).

Verificação em Tempo Polinomial

- Não se conhece nenhum algoritmo eficiente (de tempo polinomial) para resolver o problema anterior
- No entanto, dado um ciclo candidato C , é fácil verificar em tempo polinomial (linear) se cumpre a propriedade pretendida
- Neste contexto, C diz-se ser um “certificado” de uma solução (uma “prova” de que o grafo é Hamiltoniano)
- Diz-se que o problema é **verificável em tempo polinomial**, se for possível verificar em tempo polinomial se um certificado de uma solução é correto
- Nem todos os problemas têm esta característica: e.g., o problema de determinar se um grafo G tem exactamente um ciclo de Hamilton. É fácil certificar que existe pelo menos um ciclo, mas não é fácil certificar que não há mais!

Relação entre as classes P e NP

- $P \subseteq NP$
 - 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 $P \neq NP$!
 - Poder verificar se uma solução é correcta em tempo polinomial, não garante ou ajuda a encontrar um algoritmo que resolva o problema em tempo polinomial
 - Acredita-se que $P \neq NP$, mas não há provas!



Problemas NP-completos (NPC)

- A **classe de problemas NP-completos** é a classe dos problemas “mais difíceis” de resolver em toda a classe NP.
 - São pelo menos tão difíceis como qualquer outro problema em NP
- Mais precisamente, um problema de decisão A é NP-completo se (i) $A \in NP$; e (ii) qualquer problema A' em NP é **reduzível em tempo polinomial** a A ($A' \leq_p A$)
 - A redução envolve converter os dados de entrada de A' em dados de entrada de A , e os dados de saída de A em dados de saída de A'
 - As reduções serão estudadas adiante
 - Atualmente, para provar que A é NPC, basta encontrar um problema A' NPC já conhecido e provar que A' é reduzível a A em tempo polinomial
- E.g., o problema do circuito Hamiltoniano é NP-completo

Problemas NP-difíceis (NP-hard)

- Um problema NP-difícil é um problema que satisfaz a propriedade (ii) mas não necessariamente a propriedade (i)
- Ou seja, um problema de decisão A é NP-difícil se qualquer problema $A' \in NP$ é **reduzível em tempo polinomial** a A
- Por exemplo, o problema da paragem (em máquinas de Turing) é NP-difícil mas não NP-completo
 - Não pertence a NP
 - É NP-difícil, pois se pode converter qualquer problema NP no problema de paragem de uma máquina de Turing (ver referências)

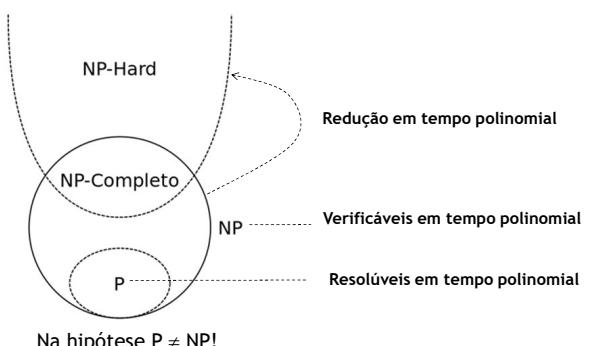
Classificação de problemas por redução

Como provar que um problema $X \in NPC$

- Provar que X está em NP
- Seleccionar um problema Y que se sabe ser NP -completo
- Definir uma redução de tempo polinomial de Y em X (*conversão de entradas*)
- Provar que, dada uma instância de Y , Y tem uma solução se, e se somente, X tem uma solução (*conversão de saídas*)



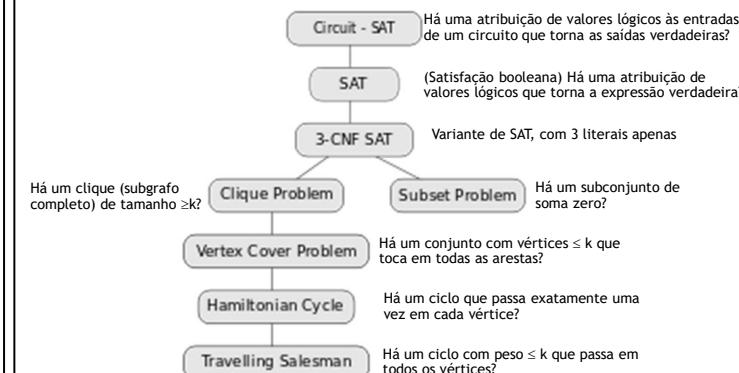
Classes P, NP, NP-complete e NP-hard



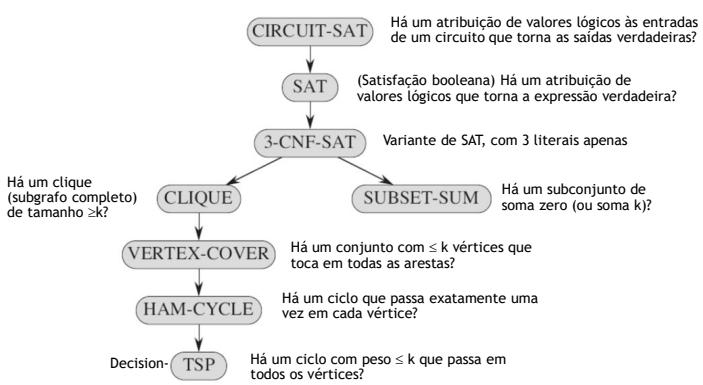
Como “provar” que um problema $X \notin P$

- Selecionar um problema Y não resolúvel em tempo polinomial ($Y \notin P$)
 - De acordo com o conhecimento atual, em que se acredita que $P \neq NP$
- Provar que Y é reduzível a X em tempo polinomial ($Y \leq_p X$)
 - Redução de entradas e saídas
 - Como a redução é efetuada em tempo polinomial, se X for resolúvel em tempo polinomial, então Y também o seria, o que contradiz a hipótese
 - Em geral, a redução de Y a X permite provar que X é pelo menos tão difícil quanto Y

Exemplos de problemas NP-completos e reduções normalmente usadas



Exemplos de problemas NPC e reduções



Redução de Problemas NP

- Definição + formal da classe dos problemas NP- completo
 - Um problema de decisão $B \in NP$ é NP-completo se $A \leq_p B \mid \forall A \in NP$
 - 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) $B \in NP$, e
 - (2) $A \leq_p B$ para algum problema A , se A é NP-Completo

Exemplos de reduções

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)

Como provar que um problema $X \in NPC$

- Provar que X está em NP
- Selecionar um problema Y que se sabe ser NP -completo
- Definir uma redução de tempo polinomial de Y em X (*conversão de entradas*)
- Provar que, dada uma instância de Y , Y tem uma solução se, e se somente, X tem uma solução (*conversão de saídas*)

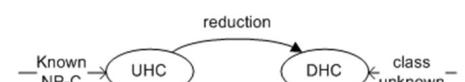


Directed Hamiltonian Cycle (DHC) é NPC?

Problema: Sabendo-se que o problema UHC (*Undirected Hamiltonian Cycle*) é NP-completo, provar que o problema DHC (*Directed Hamiltonian Cycle*) é também NP-completo

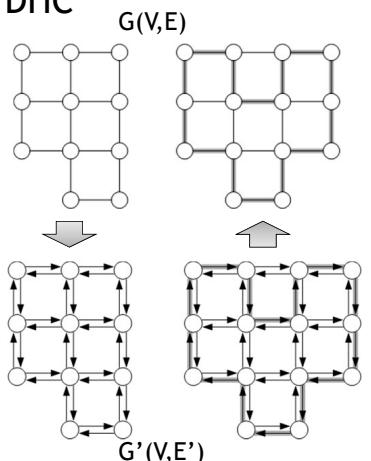
Resolução:

- Um ciclo Hamiltonian candidato é facilmente verificável em tempo polinomial, logo $DHC \in NP$
- O problema UHC é facilmente reduzível ao problema DHC em tempo polinomial (ver slide seguinte), logo $DHC \in NPC$



Redução de UHC a DHC

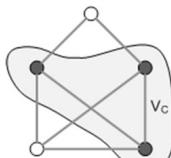
- Dado grafo não dirigido $G=(V,E)$, cria-se grafo dirigido $G'=(V,E')$ pela substituição de cada aresta $\{u, v\} \in E$ por duas arestas dirigidas (u, v) e $(v, u) \in E'$



- Cada caminho simples em G é, portanto, um caminho simples em G' , e vice-versa. Portanto, G terá um ciclo de Hamilton se, e somente se, G' também o tiver!

Vertex Cover (VC)

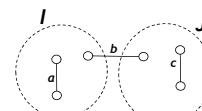
- Uma cobertura de vértices de um grafo $G = (V, E)$ é um subconjunto $V_C \subseteq V$, tal que toda aresta $(a, b) \in E$ é incidente em pelo menos um vértice $u \in V_C$.



- Vértices em V_C “cobrem” todas as arestas em G .
- Problema de decisão (VC):
 - O grafo G tem uma cobertura de vértices de tamanho $\leq k$?

Dualidade VC ↔ IS

- Dado grafo não dirigido $G=(V,E)$, seja I, J uma partição de V em dois subconjuntos disjuntos (i.e., $I \cup J = V$ e $I \cap J = \emptyset$)
- Se I é um conjunto independente de vértices, então não podem existir arestas do tipo a , logo os vértices em J tocam todos as arestas de G , logo J é uma cobertura de vértices
- Se J é uma cobertura de vértices, então não podem existir arestas do tipo a , logo I é um conjunto independente de vértices.
- I é um conj. indep. de vértices $\Leftrightarrow V \setminus I$ é uma cobertura de vértices



Redução de Problemas NP

Solução

- Considera-se q/ a rotina `bool DHC (aDigraph)` resolve o problema!

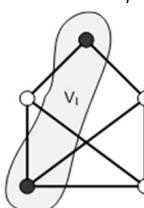
```
• Redução UHC → DHC
bool UHC (G) {
    create digraph G' with the same number of vertices as G
    foreach edge (u, v) in G
        Add edges (u, v) and (v, u) in G'
    return DHC (G')
}
```

- 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.

Independent Set (IS)

- 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 .



- Problema de decisão (IS):
 - O grafo G tem um conjunto independente de tamanho $\geq k$?

Vertex Cover é NPC?

- Problema: Sabendo-se que $IS \in NPC$, provar que $VC \in NPC$

Resolução:

- Dada um conjunto candidato de vértices V_C , é fácil verificar em tempo polinomial se $|V_C| \leq k$ e se toca em todas as arestas, logo $VC \in NP$
- Para provar que $VC \in NP-hard$, indicamos de seguida uma redução de tempo polinomial de IS em VC

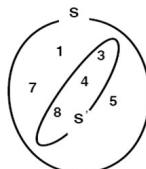


Redução de IS a VC

- Seja uma instância qualquer de IS: $G = (V, E)$, k
- Pela propriedade da dualidade, G tem um conjunto independente de vértices (V_i) de tamanho $\geq k$ se tiver uma cobertura de vértices (V_c) de tamanho $\leq k'$, com $k' = |V| - k$
- Assim, a conversão de entradas é trivial:
 - Dada uma instância qualquer de IS: $G = (V, E)$, k
 - Constrói-se uma instância de VC: $G = (V, E)$, $k' = |V| - k$
- A conversão de saídas é também trivial:
 - Conversão de 'certificados': $V_c \rightarrow V_i = V \setminus V_c$
 - Conversão de decisão: mantém-se a mesma decisão

Problema da Soma de Subconjuntos (SS)

- Dado um conjunto de inteiros positivos, S , há um subconjunto, S' em S , tal que a soma dos elementos de S' seja k ?



Ex:

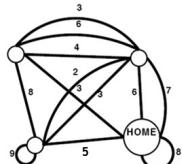
$S = \{1, 3, 4, 5, 7, 8\}$

Find S' with sum = 15!

Problema da Marcação de Exames

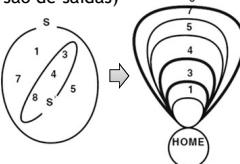
Problema do Caminhada (Jogging (J))

- Considere um grafo não dirigido G , admitindo arestas paralelas e anéis, com pesos inteiros positivos nas arestas, no qual se distingue um vértice *home*.
- O problema da caminhada (Jogging (J)) consiste em verificar se existe um caminho de peso total k , começando e terminando em *home*, sem repetir arestas.
- Prove que J é um problema NPC, sabendo-se que o problema da soma de subconjuntos é NPC.



Problema da Caminhada é NPC?

- Um caminho candidato é facilmente verificável em tempo polinomial, logo $J \in \text{NP}$
- Para provar que $J \in \text{NP-hard}$, reduz-se SS a J em tempo polinomial. Como?
 - Dado um conjunto S , cria-se um grafo G com um único vértice *home* e um anel de peso x para cada elemento $x \in S$ (conversão de entradas)
 - S tem um subconjunto de soma k se G tem um caminho de peso total k sem repetir arestas (conversão de saídas)

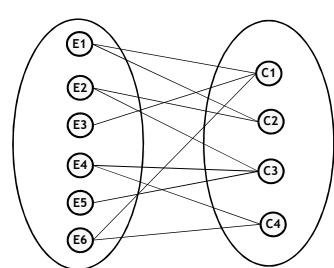


Problema da Marcação de Exames (exame 2016/17)

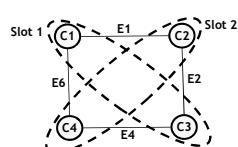
- Estudantes podem inscrever-se em vários cursos.
- Todos os exames finais terão duração de 1h.
- Determinar o número mínimo de *slots* de exame de 1 hora, a fim de evitar que estudantes inscritos em vários cursos tenham exames sobrepostos
- a) Reformule este problema como um problema de decisão.
- b) Verifique se há uma solução eficiente para este problema, explicando os passos da sua solução.
- Sugestão: Poderá utilizar as seguintes definições de problemas NP-completo: Cobertura de Vértices, Coloração de Grafos

Exemplo

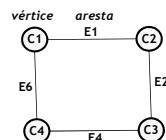
Estudantes Cursos



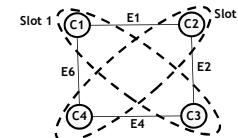
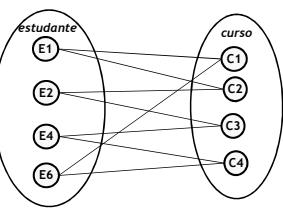
Conflitos



Coloração de Grafos



Marcação de Exames



Problema do Clique (recurso 2016/17)

- Um *clique* de um grafo não dirigido é um subconjunto dos seus vértices, tal que, para quaisquer pares de vértices u e v neste subconjunto, existe uma aresta do grafo que liga os vértices u e v . O problema de otimização consiste em encontrar um clique de tamanho máximo.
- a) Reformule este problema como um problema de decisão.
- b) Verifique se há uma solução eficiente para este problema, explicando os passos da sua solução.
- Sugestão: poderá utilizar as seguintes definições de problemas NP-completo: Coloração de Grafos, Conjunto Independente

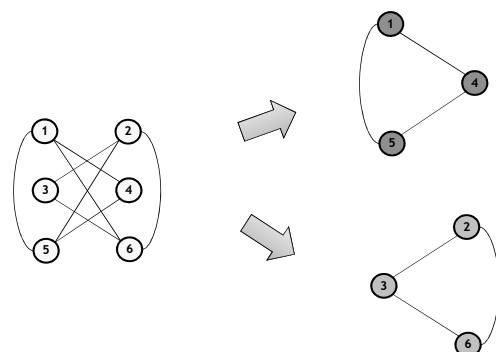
Resolução

- a) Determinar se é possível usar um número de $slots \leq k$, a fim de evitar que estudantes inscritos em vários cursos tenham exames sobrepostos
- b) O problema é NP-Completo, pois:
 - É NP, pois uma marcação candidata pode obviamente ser verificada em tempo polinomial
 - É NP-difícil, pois o problema da Coloração de Grafos é redutível em tempo polinomial ao problema da Marcação de Exames
 - Dado um grafo $G=(V,E)$, cada vértice é convertido num curso e cada aresta é convertida num estudante que está inscrito nos 2 cursos correspondentes aos vértices em que incide a aresta
 - Os *slots* da solução do problema da marcação de exames correspondem a cores no problema da coloração de grafos
 - Ver ilustrações a seguir

Problema do Clique

Exemplo

Cliques de tamanho 3:



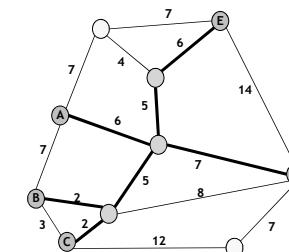
Resolução

- a) Dado um grafo não dirigido $G=(V,E)$ e um $k \in \mathbb{N}$, verificar se G tem um clique de tamanho $\geq k$
- b) O problema é NP-Completo, pois:
 - É NP, pois um clique candidato pode obviamente ser verificado em tempo polinomial
 - É NP-difícil, pois o problema do Conjunto Independente é reduzível em tempo polinomial ao problema do Clique
 - Dado um grafo não dirigido $G=(V,E)$, converte-se no grafo complementar $G'=(V,E')$ com os mesmos vértices e o conjunto complementar de arestas
 - Um clique de tamanho k de G' é um conjunto independente de tamanho k de G , e vice-versa
 - Ver ilustrações a seguir

Problema da Partilha de Viaturas

- Um grupo de pessoas pretende organizar os transportes em viatura própria (ida e regresso) para uma atividade de lazer num ponto definido, minimizando o consumo de combustível.
 - Assumir que o consumo depende apenas da distância percorrida
- Para esse efeito, pessoas partindo de casas diferentes nas suas viaturas podem encontrar-se em pontos intermédios, deixando aí um dos carros (procedendo de forma inversa no regresso).
 - Assumir que é possível deixar o carro em qualquer ponto
- Mostrar que é um problema NP-completo, sabendo-se que o problema da Árvore de Steiner em Grafos (ver slide seguinte) é NP-completo

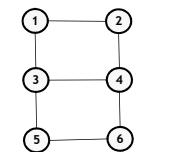
Problema da Árvore de Steiner (2/2)



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

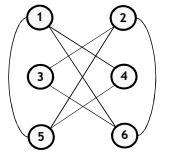
Peso total da árvore: 33

Problema do Conjunto Independente



(conjunto independente de tamanho 3)

Problema do Clique



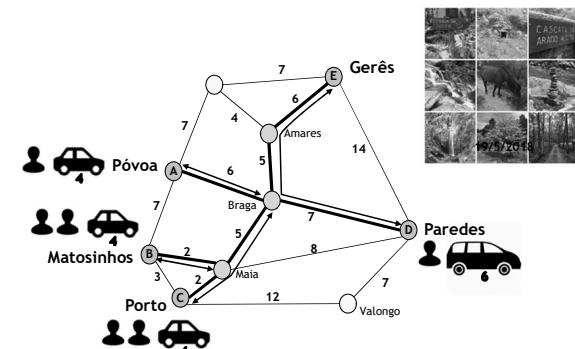
(clique de tamanho 3)

(complemento)

Problema da Árvore de Steiner (1/2)

- Seja $G = (V, E)$ um grafo não dirigido com pesos não negativos
- Seja $S \subseteq V$ um subconjunto de vértices, chamados **terminais**.
- Uma **árvore de Steiner** é uma árvore em G que contém todos os vértices de S .
- Problema de otimização:** encontrar uma árvore de Steiner de peso mínimo
 - É o mesmo que uma árvore de expansão mínima, no caso de $S = V$
- Problema de decisão (com pesos inteiros):** determinar se existe uma árvore de Steiner de peso total que não excede um número natural k pré-definido
 - Sabe-se que é um problema NP-completo

Problema da Partilha de Viaturas



Distância total percorrida pelo conjunto de viaturas: $2 \times (18 + 6 + 2 + 7) = 2 \times 33 = 66$

Resolução

- Problema de decisão: é possível efetuar o transporte com distância total percorrida $\leq k$, pelo conjunto de viaturas?
- É um problema NP, pois uma solução candidata (com plano de percursos das viaturas) pode ser facilmente verificada em tempo polinomial
- É um problema NP-difícil, pois o problema da Árvore de Steiner é redutível ao problema da Partilha de Viaturas em tempo polinomial
 - Faz-se corresponder conjunto S a conjunto de pontos de partida e de chegada (pode-se escolher um arbitrariamente como ponto de chegada)
 - Cada ponto de partida tem uma pessoa e uma viatura de capacidade=| S |
 - Existe árvore de Steiner de peso total $\leq k$ se e só se existe forma de partilha de viaturas com peso total (distância total) $\leq 2k$