

## Algoritmos Gananciosos (Greedy)

- Aplica heurística de solução, realizando uma escolha ótima local.
- Aplicável a problemas de otimização
- Em diversos problemas, a otimização local garante a otimização global ← sub Estrutura ótima.
- Características: conjunto de candidatos; função de seleção que escolhe o melhor candidato a ser incluído na solução; função de viabilidade que determina se o candidato pode ou não fazer parte da solução; função objetivo que atribui um valor a uma solução; função solução que determina se chegou a solução completa

## Troco de moedas

- Sistema de moedas canónico: algoritmo gananciosa encontra sempre uma solução ótima para o problema do troco (com stock ilimitado)
- Extrair a moeda de valor mais alto que não excede o montante em falta.
- Sendo  $C = \{c_1, c_2, \dots, c_n\}$  as denominações do sistema de moedas, se o sistema não for canónico, o menor contra exemplo situa-se na gama  $c_3 + 1 < x < c_{n-1} + c_n$ , basta fazer pesquisa exaustiva nessa gama para determinar se é canónico.

## Escalonamento de atividades

- **Problema:** dado um conjunto de atividades, encontrar um subconjunto com o maior numero de atividades não sobrepostas
- **Input:** conjunto A de n atividades,  $a_1, a_2, \dots, a_n$  com  $s_i$  = instante de inicio e  $f_i$  = instante de fim
- **Ouptut:** subconjunto R com o numero máximo de atividades compatíveis.
- Ordenar as atividades numa ordem especifica (fim mais cedo ou inicio mais tardio); escolher a melhor opção; descartar as incompatíveis com a escolhida; proceder da mesma forma.
- Sendo A – conjunto inicial de atividades;  $a$  – atividade selecionada com fim mais cedo; I – conjunto de atividades incompatíveis com  $a$ ; C – conjunto de atividades Restantes. Do conjunto  $\{a\} \cup I$  só pode ser selecionada no máximo uma atividade pois são mutuamente incompatíveis, que com outro critério de ordenação poderia haver mais. Desse conjunto escolhemos uma, que é o máximo possível. A atividade escolhida não tem incompatibilidade com as restantes logo a escolha de  $a$  permite maximizar o nº de atividades que se podem escolher de C.
- **Problema 2:** sequenciar tarefas minimizando o tempo médio de conclusão. Método de ordenação usado é escolher tarefas mais curtas primeiro.

## Backtracking

- Algoritmos de tentativa e erro.
- Explorar um espaço de estados a procura de um estado-objetivo;
- Ao chegar a um ponto de escolha, escolher uma das opções; chegando a um “beco sem saída”, retroceder ate ao ponto de escolha mais próximo com alternativas por explorar, tentar outra alternativa.
- Problema do troco com limitações de stock; sudoku; 8 rainhas; labirintos.
- Uma representação é árvore de espaço de estados: A raiz representa o inicio (0 escolhas); Nós ao nível 1 representam a primeira escolha; caminhos da raiz até às folhas representam soluções candidatas.
- Outra representação é uma árvore n-ária em que, em cada nível, se escolhe o próximo valor a incluir. Cada nó representa uma solução candidata (total  $2^n$  )
- Tempo de execução no pior caso (pesquisa exaustiva do espaço de estados) é determinado pela dimensão do espaço de estados, que muitas vezes é exponencial.
- Variantes: Encontrar uma solução; encontrar todas as soluções (não para a exploração); encontrar a melhor solução (variante de encontrar todas as soluções com possibilidade de pruning quando a solução atual não leva a uma melhor do que as já encontradas).

## Pruning

- Interromper (podar) a pesquisa e retroceder em nos que garantidamente não levam a uma solução viável (chamados nos não promissores)

## Soma de Subconjuntos

- **Problema:** dado um conjunto (ou multi conjunto)  $W = \{w_1, \dots, w_n\}$  de inteiros positivos e soma S a perfazer, encontra um subconjunto R de W com soma S.
- Uma possibilidade é uma árvore binária em que, em cada nível k, se decide da inclusão ou não do valor  $w_k$ . As folhas representam as soluções candidatas (soma das inclusões).
- As folhas da árvore binária representam os possíveis subconjuntos de W, em número  $2^n$ . O numero de nos da árvore é sensivelmente o dobro  $2^{(n+1)} - 1$ . No pior caso  $T(n) = O(2^n)$ .
- **Pruning:** a soma já selecionada é superior à soma a perfazer ou a soma ainda selecionável é inferior à soma a perfazer.

## Divide And Conquer

- **Dividir** o problema em subproblemas que são instâncias mais pequenas do mesmo problema. **Conquistar** os subproblemas resolvendo-os recursivamente ou diretamente. **Combinar** as soluções.
- Subproblemas devem ser disjuntos (usar programação dinâmica). Dividir em subproblemas de dimensão similar para maior eficiência. Para existir divisão devem existir 2 ou mais chamadas recursivas.
- Normalmente desempenho ótimo com  $n$  threads =  $n$  cores, hyper threading  $n$  ótimo é  $2 * n$  cores.

## Merge-Sort

- Ordenar 2 subsequências de igual dimensão e juntá-las.
- $T(n)=O(n \cdot \log(n))$  tanto no pior caso como no caso médio.
- $S(n)=n$
- Otimizar com memória auxiliar, insertion sort com  $n < 20$ , processamento paralelo.

## Quicksort

- Ordenar elementos menores e maiores que pivot, concatenar.
- $T(n)=O(n^2)$  no pior caso (1 elemento menor, restantes maiores)
- $T(n)=O(n \cdot \log(n))$  no melhor caso e no caso médio (com escolha aleatória do pivot)
- $S(n)=1$

## Calculo de $x^n$

- Resolução iterativa com  $n$  multiplicações:  $T(n)=O(n)$

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

- Resolução com divisão e conquista com número de multiplicações reduzido para  $\log(n)$
- $T(n)=O(\log(n))$   $S(n)=O(\log(n))$
- Classificação como divisão e conquista não é consensual por apenas haver uma chamada recursiva (subproblemas idênticos).

## Pesquisa binária (Binary Search)

- $T(n)=O(\log(n))$
- Classificação como divisão e conquista não consensual por um dos 2 subproblemas ser vazio.

# Dynamic Programming

- Problemas resolúveis recursivamente (solução é uma combinação de soluções de subproblemas similares), mas que resolução direta duplicaria trabalho com resoluções repetidas dos mesmos subproblemas.
  - Duas abordagens: economizar tempo (evitar repetir trabalho) memorizando as soluções parciais dos subproblemas (gastando memoria); economizar memoria, resolvendo subproblemas por ordem que minimiza o numero de soluções parciais a memorizar (bottom-up, começando pelos casos base, mais simples).

# Cálculo de número de combinações $n_{C_k}$

${}^nC_k$	k=0	k=1	K=2	K=3	K=4	k=5
n=0	1					
n=1	1	1				
n=2	1	2	1			i
n=3	1	3	3	1		
n=4	j	4	6	4	1	
n=5	1	5	10	10	5	1

- No exemplo  $n=5$  e  $k=2$
  - Abordagem bottom-up. Apenas necessário manter uma lista de resultados no percurso da solução. Começar na primeira coluna com  $k = 0$ , e continuar até chegar à ultima coluna. Usado para conservar **memória**.
  - $j_{max} = n - k, 0 \leq j \leq j_{max}$  logo  $j_{max} = 3, 0 \leq j \leq 3$
  - $T(n, k) = O(k \cdot (n - k))$
  - $S(n, k) = O(n - k)$

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

# Problema da mochila

- **Problema:** Encontrar combinação de itens de vários tamanhos e valores que maximiza o valor total guardando-os numa mochila de capacidade limitada. Assumindo capacidades inteiros e numero de itens ilimitado.
  - Calcular a melhor combinação para todas as mochilas de capacidade 1 até  $M$  (capacidade da mochila). Começar por considerar que só se pode usar o item 1 (primeira iteração), depois os itens 1 e 2, etc., e finalmente todos os itens de 1 a  $N$  (numero de itens) (ultima iteração).
  - **Dados:**  $N$  – numero de itens (com numero 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.
  - **Dados de trabalho, no fim 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  $i$ ;  $\text{best}[k], i \leq k \leq M$  - último

$\text{best}[k]^{(i)} = 0$

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

$$\text{cost}[k]^{(i)} = \begin{cases} \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)}, & \text{se } \begin{cases} \text{size}[i] \leq k \\ \text{val}[i] + \text{cost}[k - \text{size}[i]]^{(i)} > \text{cost}[k]^{(i-1)} \end{cases} \\ \text{cost}[k]^{(i-1)}, & \text{no caso contrário} \end{cases}$$

$$\text{best}[k]^{(i)} = \begin{cases} i, & \text{no primeiro caso acima (usa o item } i\text{)} \\ \text{best}[k]^{(i-1)}, & \text{no segundo caso acima (não usa o item } i\text{)} \end{cases}$$

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

item selecionado para obter o melhor valor com mochila de capacidade k, usando apenas itens de 1 a i.

- **Dados de saída:**  $\text{cost}[M]$  - melhor valor que se consegue com mochila de capacidade M;  $\text{best}[M], \text{best}[M - \text{size}[\text{best}[m]]]$  - itens selecionados.
- $T(N, M) = O(N \cdot M)$      $S(N, M) = O(M)$

## Números de Fibonacci

- Memorizar os dois últimos elementos da sequência para calcular o seguinte.

## Subsequência crescente mais comprida

- Exemplo: Sequencia S = (9, 5, 2, 8, 7, 3, 1, 6, 4); Subsequência crescente mais comprida (elementos não necessariamente contíguos): (2, 3, 4) ou (2, 3, 6).
- Formulação:  $s_1, \dots, s_n$  - sequência,  $l_i$  - comprimento da maior subsequência crescente de  $(s_1, \dots, s_i)$ ,  $p_i$  - predecessor de  $s_i$  nessa subsequência crescente.
- $l_i = 1 + \max\{l_k | 0 < k < i \wedge s_k < s_i\}$      $p_i$  = valor de k escolhido para o máximo na expressão anterior.
- Comprimento  $\max(l_i)$

	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

- $T(n) = O(n^2)$      $S(n) = O(n)$

## Troco de moedas

Consiste em escolher usar a moeda em i ou não usar e escolher a moeda em i-1

	k=0	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8=m
i=0 v0=- C <sub>0,k</sub>	0	-	-	-	-	-	-	-	-(ou $\infty$ )
	-	-	-	-	-	-	-	-	-(ou 0)
i=1 v1=1 C <sub>1,k</sub>	0	1	2	3	4	5	6	7	8
	-	1	1	1	1	1	1	1	1
i=2 v2=4 C <sub>2,k</sub>	0	1	2	3	1	2	3	4	2
	-	1	1	1	2	2	2	2	2
i=3 v3=5 C <sub>2,k</sub>	0	1	2	3	1	1	2	3	2
	-	1	1	1	2	5	5	5	2

- $T(n, m) = O(n \cdot m)$
- $S(n, m) = O(m)$
- m – valor a trocar.
- v<sub>0</sub>, ..., v<sub>n</sub> – moedas.
- n – numero de moedas.
- $C_{i,k}$  – número mínimo de moedas que se consegue trocar k com moedas até i.
- $P_{i,k}$  - valor da última moeda usada para alcançar o mínimo.

- $C_{i,0} = 0$ ;     $C_{0,k} = \infty$  (se  $k > 0$ );     $P_{0,k} = P_{i,0} = \text{indefinido}$  (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:  $v_{P_{n,m}}, v_{P_{n,m}-v_{P_{n,m}}}, \dots$

# Correção de Algoritmos

	Análise estática (teórica)	Análise dinâmica (experimental)	
Eficiência temporal e espacial	complexidade assintótica	testes de desempenho; <i>profiling</i>	<ul style="list-style-type: none"> <li>Para provar que um algoritmo resolve corretamente um problema, precisamos de uma especificação rigorosa do problema e de uma descrição rigorosa do algoritmo.</li> </ul>
Funcionamento correto	prova ou argumentação sobre correção	testes pontuais ou aleatórios (*)	<ul style="list-style-type: none"> <li><b>Entradas:</b> Dados de entrada e restrições associadas (<b>pré-condições</b>)</li> </ul>

- Saídas:** Dados de saída e restrições associadas (**pós-condições**), sendo que objetivos de maximização e minimização podem ser reduzidos a restrições.
- Correção parcial:** se o algoritmo for executado com entradas que obedecem às pré-condições, então, se terminar, produz **saídas corretas**, ou seja, obedecem às pós-condições.
- Correção total:** se o algoritmo for executado com entradas que obedecem às pré-condições, então termina, produzindo saídas que obedecem as pós-condições.

## Square Root

- Pré-condições:  $x \geq 0$  ; Pós-condições:  $RESULT \cdot RESULT = X \wedge RESULT \geq 0$

## Binary Search

- Pré-condições: Array ordenado;  $a \neq NULL$  ; Operadores de ordenação definidos para o tipo T; Pós-condições:  $(0 \leq RESULT < n \wedge a[RESULT] = x) \vee (RESULT = -1 \wedge x \notin a)$

## 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** que é um a expressão booleana sempre verdadeira ao longo do ciclo e mostrar que é verdadeira **inicialmente** ou seja é implicada pela pré-condição; é **mantida** em cada iteração, ou seja, é verdadeira no fim da 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**.

## Insertion Sort

### Invariante do ciclo principal [ I(j) ] ?

- $A[1, \dots, j-1]$  contém os elementos originais, mas ordenados ( $j = 2, \dots, n + 1$ ).
- É valido **inicialmente** ( $j = 2$ ), pois 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 algoritmo 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-se a pós-condição: Invariante refere-se a  $A[1..n]$  ou seja toda a array; Logo, implica trivialmente a pós-condição, pois é coincidente.

### Variante do ciclo principal [ v(j) ] ?

- $n+1-j, (j=2, \dots, n+1)$
- 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).

## Binary Search

### Invariante do ciclo principal [ I(low, high) ] ?

- $x$  só pode existir na área de pesquisa entre  $low$  e  $high$
- É valido **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  $< x$ . Não se exclui nunca o  $x$ .
- No **fim do ciclo** garante-se 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  $A$ .

### Variante do ciclo principal [ v(low, high) ] ?

- $high - low + 1 \leftarrow$  largura da área de pesquisa
- Inteiro, pois  $low$  e  $high$  são inteiros; Não negativo, pois no pior caso  $low = high$ ; Estritamente decrescente, pois em cada iteração ou aumenta-se o  $low$  ou diminui-se  $high$ .
- Logo tem correção total.

# Graph Theory

**Grafo**  $G=(V,E)$

- $V$  – conjunto de **vértices** (ou nós);  $E$  – conjunto de **arestas** (ou arcos); cada aresta é um par de vértices  $(v,w) \quad v,w \in V$ ; se o par for ordenado, o grafo é dirigido, ou **digrafo**; um vértice  $w$  é **adjacente** a um vértice  $v$  se e só se  $(v,w) \in E$ ; num grafo não dirigido com aresta  $(v,w)$  e logo  $(w,v)$ ,  $w$  é adjacente a  $v$  e  $v$  adjacente a  $w$ .

## Caminhos

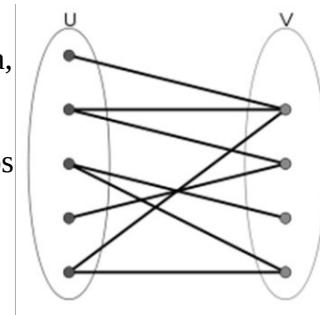
- **Caminho** – sequência de vértices  $v_1, \dots, v_n$  tais que  $(v_i, v_{i+1}) \in E, 1 \leq i < n$ ; comprimento do caminho é o número de arestas,  $n - 1$ ; se  $n = 1$ , caminho reduz-se a 1 vértice, comprimento 0;
- **Caminho simples** – todos os vértices distintos, exceto possivelmente o primeiro e o último.

## Ciclos

- **Ciclo (ou circuito)** – caminho de  $\text{comprimento} \geq 1$  com  $v_1 = v_n$ ; Num grafo não dirigido, requer-se que as arestas sejam diferentes
- **Anel** – caminho  $v, v \Rightarrow (v, v) \in E$ , comprimento 1; raro

## Tipos de grafos

- **Grafo acíclico dirigido (DAG – Directed Acyclic Graph)** - Grafo dirigido sem ciclos. Para qualquer vértice  $v$ , não há nenhuma ligação dirigida começando e acabando em  $v$ .
- **Grafo simples** - Grafo sem arestas paralelas (várias adjacências, para o mesmo par de vértices) nem anéis.
- **Grafo pesado** – As arestas são etiquetadas com um peso (distância, custo, ...)
- **Grafo bipartido** – Conjunto de vértices é partido em dois subconjuntos  $V_1$  e  $V_2$ . Arestas ligam vértices de diferentes partições.



## Conektividade

- Grafo não dirigido é **conexo** se e só se houver um caminho a ligar qualquer par de vértices.
- Digrafo com a mesma propriedade: **fortemente conexo**, se para 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 (correspondente) é conexo.

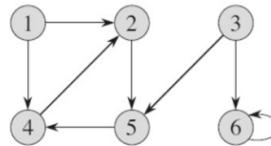
## Densidade

- **Denso** -  $|E| \sim O(V^2)$  ; **Esparsos**  $|E| \sim O(V)$

# Representações

## Matriz de adjacências

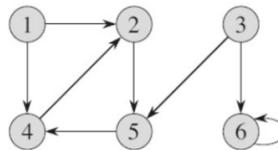
- Apropriada para grafos densos
- Elementos da matriz podem ser os pesos
- Grafos não dirigidos: matriz simétrica.



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

## Lista de adjacências

- Espaço é  $O(|E|+|V|)$
- Pesquisa de adjacentes em tempo proporcional ao número destes.
- Estrutura típica para grafos esparsos.
- Grafos não dirigidos: lista com dobro do espaço.



1	2	-	4	/
2	5	/		
3	6	-	5	/
4	2	/		
5	4	/		
6	6	/		

## Pesquisa em profundidade (depth-first)

- Areias são exploradas a partir do vértice v mais recentemente descoberto que ainda tenha arestas a sair dele.
- Quando todas as arestas de v forem exploradas, retorna para explorar arestas que saíram do vértice a partir do qual v foi descoberto.
- Se se mantiverem vértices por descobrir, um deles é selecionado como a nova fonte e o processo de pesquisa continua a partir dai.
- Todo o processo é repetido ate todos os vértices serem descobertos.
- Deteção de ciclos: quando um vértice adjacente está na stack de visita.
- No pior caso:  $T(V, E) = O(|V| + |E|)$      $S(V, E) = O(|V|)$

## Pesquisa em largura (breadth-first)

- Dado um vértice fonte s, explora-se sistematicamente o grafo descobrindo todos os vértices a que se pode chegar a partir de s (vértices adjacentes)
- Só depois é que se passa para outro vértice.
- Cria árvore de expansão em largura, com raiz s.
- 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)
- No pior caso:  $T(V, E) = O(|V| + |E|)$      $S(V, E) = O(|V|)$

# Ordenação Topológica

## Problema

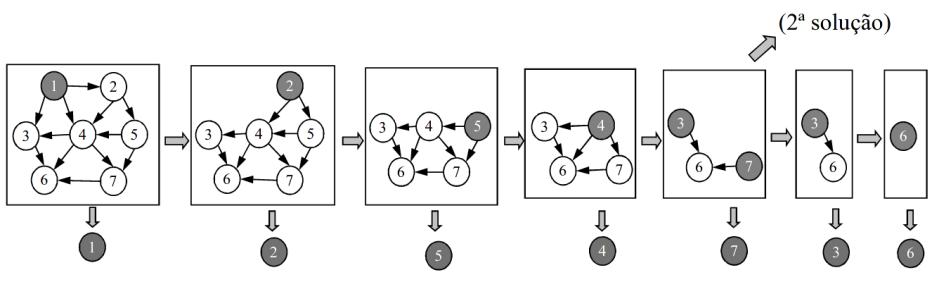
- Ordenar os vértices de um DAG tal que, se existe uma aresta  $(v, w)$  no grafo, então v aparece antes de w. Intuitivamente, dispor as setas todas no mesmo sentido; Impossível se o grafo for cíclico; Pode existir mais do que uma ordenação.

## Método baseado em DFS

- Na DFS de um DAG, a pós-ordem de visita dá uma ordenação topológica inversa. No entanto o método não é genérico, pois algumas ordenações topológicas válidas não podem ser obtidas desta forma.

## Método geral

- Descobrir um vértice sem arestas de chegada (indegree = 0); Imprimir/Guardar o vértice; Eliminá-lo, assim como as arestas que dele saem; Repetir o processo no grafo restante.



## Refinamento da ordenação

- Simular eliminação atualizando indegree (número de arestas que chegam a v, partindo de vértices por visitar) dos vértices adjacentes; Memorizar numa estrutura auxiliar vértices por imprimir com indegree = 0 (C - conjunto de vértices por visitar cujo indegree é 0).
- Se o resultado tiver um número de vértices diferente do grafo original, então o grafo tem ciclos.

## Análise do algoritmo

- Se as inserções e eliminações em C forem efetuadas em tempo constante, o algoritmo pode ser executado em tempo  $O(|V|+|E|)$ ; o corpo do ciclo de atualização de indegree é executado no máximo uma vez por aresta; as operações de inserção e remoção na fila são executados no máximo uma vez por vértice; a inicialização leva um tempo proporcional ao tamanho do grafo.

## Shortest path

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

## Variantes

- Caso base: grafo dirigido, fortemente conexo, pesos  $\geq 0$

- **Grafo não dirigido:** Mesmo que grafo dirigido com pares de arestas simétricas.
- **Grafo não conexo:** Pode não existir caminho para alguns vértices, ficando distância infinita
- **Grafo não pesado:** Mesmo que peso 1 (mais curto = com menos arestas); Existe algoritmo mais eficiente para este caso do que para caso base.
- **Arestas com pesos negativos:** Existe algoritmo menos eficiente para este caso do que para o caso base; Ciclos com peso negativo tornam o caminho mais curto indefinido.

## Grafo dirigido não pesado (BFS)

- Método básico: pesquisa em largura (BFS) + cálculo de distâncias:
- Marcar o vértice  $s$  com distância 0 e todos os outros com distância infinita; Entre os vértices já alcançados (distância  $\neq$  infinito) e não processados (no próximo passo), escolher para processar o vértice  $v$  marcado com distância mínima; Processar vértice  $v$ : analisar os adjacentes de  $v$ , marcando os que ainda não tinham sido alcançados (distância infinita) com distância  $v + 1$ ; Voltar ao passo 2, se existirem mais vértices para processar.
- Usando uma fila (FIFO) para inserir os novos vértices alcançados e extrair o próximo vértice a processar, garante-se a ordem de progressão pretendida.
- Associa-se a cada vértice a seguinte informação:  $dist$  - distância ao vértice inicial (definida/definitiva ao alcançar um vértice pela primeira vez; ao alcançar um segundo caminho, distância nunca diminui);  $path$  - vértice antecessor no caminho mais curto.
- No pior caso:  $T(V, E)=O(|V|+|E|)$      $S(V, E)=O(|V|)$

## Grafo dirigido pesado (pesos $\geq 0$ ) (Dijkstra)

- Método básico semelhante ao caso do grafo não pesado; Distância obtém-se somando pesos das arestas em vez de 1.
- Próximo vértice a processar continua a ser o de distância mínima mas já não é necessariamente o mais antigo o que obriga a usar uma fila de prioridades (com mínimo à cabeça) em vez de uma fila simples. No entanto pode ser necessário rever a distância de um vértice alcançado e ainda não processado (vértice na fila) o que obriga a usar uma fila de prioridades alteráveis (com uma implementação de DECREASE-KEY). Isto é necessário para garantir que a distância ao vértice de partida dos vértices já processados não é mais alterada.
- É um algoritmo **ganancioso**: em cada passo procura maximizar o ganho imediato (neste caso, minimizar a distância).
- No pior caso:  $T(V, E)=O((|V|+|E|)\cdot \log |V|)$  ou  $T(V, E)=O(|V|+|E|\cdot \log |V|)$  com o uso de Fibonacci heap;  $S(V, E)=O(|V|)$
- $O(|V|\cdot \log |V|)$  na extração e inserção na fila de prioridades;  $O(|E|\cdot \log |V|)$  - DECREASE-KEY, feito no máximo  $|E|$  vezes (uma vez por cada aresta).

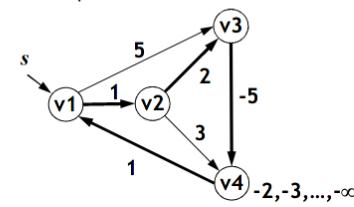
## Eficiência de DECREASE-KEY

- Fila de prioridades implementada com um heap
- Método naïve  $O(n)$ : Procurar sequencialmente no array o objeto cuja chave se quer alterar
- Método melhorado  $O(\log n)$ : Cada objeto colocado no heap guarda a sua posição (índice) no array, logo não é necessário procurar o objeto no array; Introduz um overhead mínimo nas inserções e eliminações (atualizar o índice no objeto).
- Método otimizado:  $O(1)$  com o uso de Fibonacci heaps.

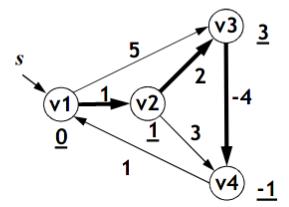
## Arestas com peso negativo sem ciclos negativos (Bellman-Ford)

- Pode ser necessário processar cada vértice mais do que uma vez. Se existirem ciclos com peso negativo, o problema não tem resolução. Não existindo ciclos com peso negativo, o problema é resolúvel em tempo  $O(|E| \cdot |V|)$  pelo algoritmo de Bellman-Ford.

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.



- Em cada iteração  $i$  ( $i$  de 1 até  $|V|-1$ ), 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**). Como no máximo 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.
- É um caso de **programação dinâmica** (utilização de resultados anteriores, casos mais simples).

## Grafos acíclicos

- Simplificação do algoritmos 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
- $T(V, E) = O(|V| + |E|)$

## Caminho mais curto entre dois vértices

- Não se conhece algoritmo mais eficiente a resolver este problema do que a resolver o mais geral (de um vértice para todos os outros).

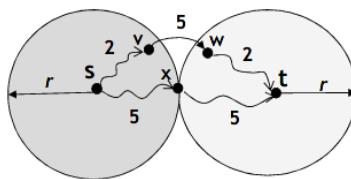
- Otimização: parar assim que chega a vez de processar o vértice de destino.

## Dijkstra

- Uma vez que o algoritmo processa os vértices por distâncias crescentes ao vértice de partida, é inspecionado um círculo em torno de  $s$  de raio igual à distância entre  $s$  e  $t$ .

## Pesquisa bidirecional

- Executar o algoritmo de Dijkstra no sentido de  $s$  para  $t$  e em sentido inverso de  $s$  para  $t$  (no grafo invertido), alternando entre um e outro. Terminar quando se vai processar um vértice  $x$  já processado na outra direção (podendo o caminho mais curto passar por  $x$  ou não).
- Manter a distância  $\mu$  do caminho mais curto conhecido entre  $s$  e  $t$ : ao processar uma aresta  $(v, w)$  tal que  $w$  já foi processado na outra direção, verificar se o correspondente caminho  $s-t$  melhora  $\mu$  (ao processar o vértice  $v$ , logo não termina).
- Retornar a distância  $\mu$  e o caminho correspondente.



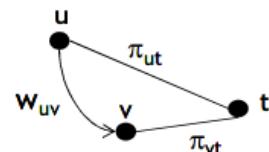
Área processada  $\sim 2\pi r^2$ , em vez de  $\sim 4\pi r^2$  na pesquisa unidirecional.

Ganho (speedup)  $\sim 2x$

## Pesquisa orientada

- 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);  $\pi_{vt}$  - **estimativa por baixo da distância mínima de  $v$  a  $t$**  (função potencial).
- Em geral, não garante o ótimo. Em certos casos, garante o ótimo, por exemplo quando pesos em arestas são distâncias em km e  $\pi_{vt}$  é a distância Euclidiana (em linha reta) de  $v$  a  $t$ .
- Equivale a aplicar o algoritmo de Dijkstra com pesos das arestas modificados  $w'_{uv} = w_{uv} - \pi_{ut} + \pi_{vt}$ , somando-se no final  $\pi_{st}$ .
- Pode ser combinado com pesquisa bidirecional.

- 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  $(s, v_1, v_2, \dots, v_k)$ , 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)

- Pré-processamento decompõe a rede em vários níveis hierárquicos (local, highway, super-highway)

## Caminho mais curto entre todos os pares de vértices

- Execução repetida do algoritmo de Dijkstra (ganancioso)  $O(|V| \cdot (|V| + |E|) \cdot \log |V|)$  - bom se o grafo for esparso  $|E| \sim |V|$ .
- **Algoritmo de Floyd-Warshall**, programação dinâmica:  $O(|V|^3)$ 
  - Melhor que o anterior se o grafo for denso  $|E| \sim |V|^2$ ; Baseia-se em **matriz de adjacências**  $W[i, j]$  com pesos.
  - Invariante: 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 ( $k=0$ ):**

$$D[i, j]^{(0)} = W[i, j] \quad P[i, j](0) = \text{nil}$$

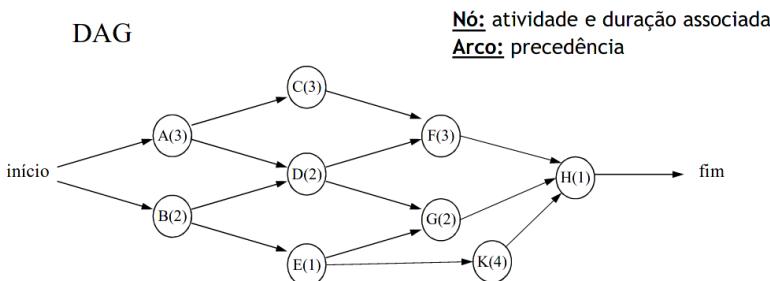
- **Recorrência ( $k=1, \dots, |V|$ ):**

$$D[i, j]^{(k)} = \min(D[i, j]^{(k-1)}, D[i, k]^{(k-1)} + D[k, j]^{(k-1)})$$

- Valor de  $P[i, j]^{(k)}$  é atualizado conforme o termo mínimo escolhido

## Aplicação a gestão de projetos

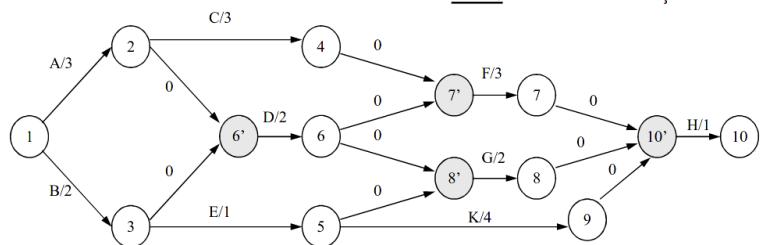
### Grafo Nó-Atividade / Grafo Nó-Evento



Qual a duração total mínima do projeto?

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

Nó: evento - completar atividade  
Arco: atividade e duração

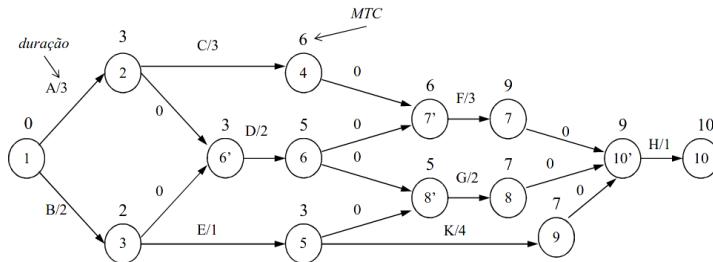


Introduzem-se nós e arcos extra para garantir precedências no caso de atividades com mais que uma antecessora

## Menor Tempo de Conclusão

- Caminho mais comprido do evento inicial ao nó de conclusão da atividade: adaptar algoritmo de caminho mais curto para grafos acíclicos (ordem topológica).

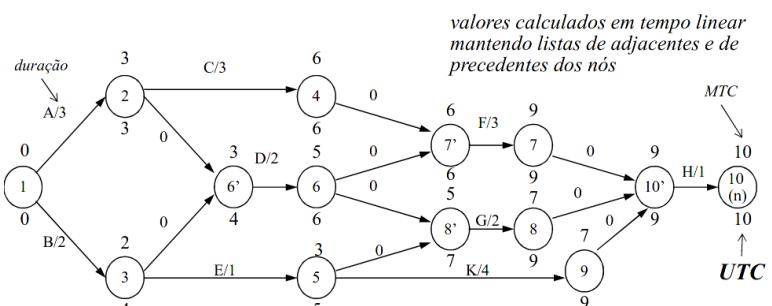
$$\bullet \quad MTC(1)=0 ; \quad MTC(w)=\max\{MTC(v)+c(v,w) \mid (v,w) \in E\}$$



## Último Tempo de Conclusão

- O tempo mais tarde que uma atividade pode terminar sem comprometer as que se lhe seguem (usando uma ordem topológica inversa)

$$\bullet \quad UTC(n)=MTC(n) ; \quad UTC(v)=\min\{UTC(w)-c(v,w) \mid (v,w) \in E\}$$



## Folga nas atividades

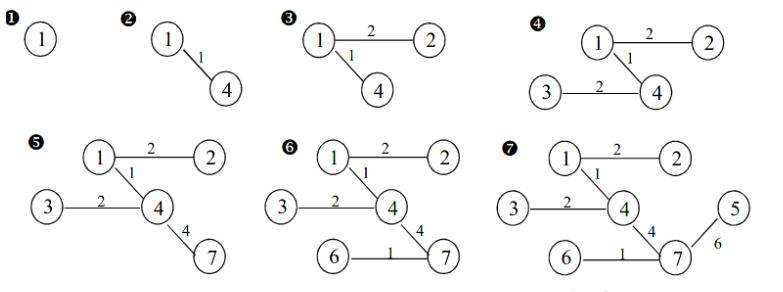
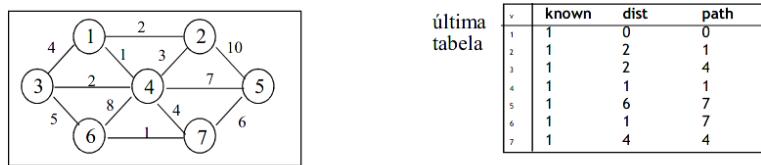
- $folga(v,w)=UTC(w)-MTC(v)-c(v,w)$
- Caminho Crítico:** só atividades de folga nula (há pelo menos 1).

## Árvore de expansão mínima (minimum spanning trees)

- Árvore que liga todos os vértices do grafo usando arestas com um custo total mínimo. É um grafo conexo acíclico, com um número de arestas =  $|V|-1$
- Só pode ser aplicado a grafos não dirigidos. O grafo tem de ser conexo.
- Minimizar o tamanho total da rede.

## Algoritmo de Prim

- Expandir a árvore por adição sucessiva de arestas e respetivos vértices. Critério de seleção: escolher a aresta  $(u, v)$  de menor custo tal que  $u$  já pertence à árvore e  $v$  não (ganancioso). Tem inicio num vértice qualquer.
- Idêntico ao algoritmo de Dijkstra para o caminho mais curto. Para cada vértice é guardado o custo mínimo das arestas que ligam a um vértice já na árvore -  $dist(V)$  ; o ultimo



vértice a alterar  $dist(V)$  -  $path(V)$  ; um indicador se o vértice já foi processado, ou seja, se já pertence à árvore -  $known(V)$

- Após a seleção do vértice  $v$ , para cada  $w$  não processado, adjacente a  $v$ ,  $dist(w)=\min\{dist(w), cost(v,w)\}$  ao invés de  $dist(w)=\min\{dist(w), dist(V)+cost(v,w)\}$
- $O(|V^2|)$  - sem fila de prioridade;  $O(|E|\cdot \log|V|)$  - com fila de prioridade.

## Algoritmo de Kruskal

- Analisar as arestas por ordem crescente de peso e aceitar as que não provocarem ciclos (ganancioso). 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.
- Aceitar arestas implica utilizar algoritmos de união e de procura de conjuntos disjuntos que são representados como árvores. Se dois vértices pertencem à mesma árvore/conjunto, mais uma aresta entre eles provoca um ciclo. Se são de dois conjuntos disjuntos, aceitar a aresta é aplicar uma união.
- A seleção de arestas é melhor feita com uma fila de prioridade em tempo linear e ordenar pelo peso.
- No pior caso  $O(|E|\cdot \log|E|)$  devido às operações na fila e como  $|E|\leq V^2$ ,  $\log|E|\leq 2\cdot \log|V|$  , logo a eficiência também é  $O(|E|\cdot \log|V|)$  .

```

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);
    }
}

```

## Coneectividade (Connectivity)

### Grafo não dirigido

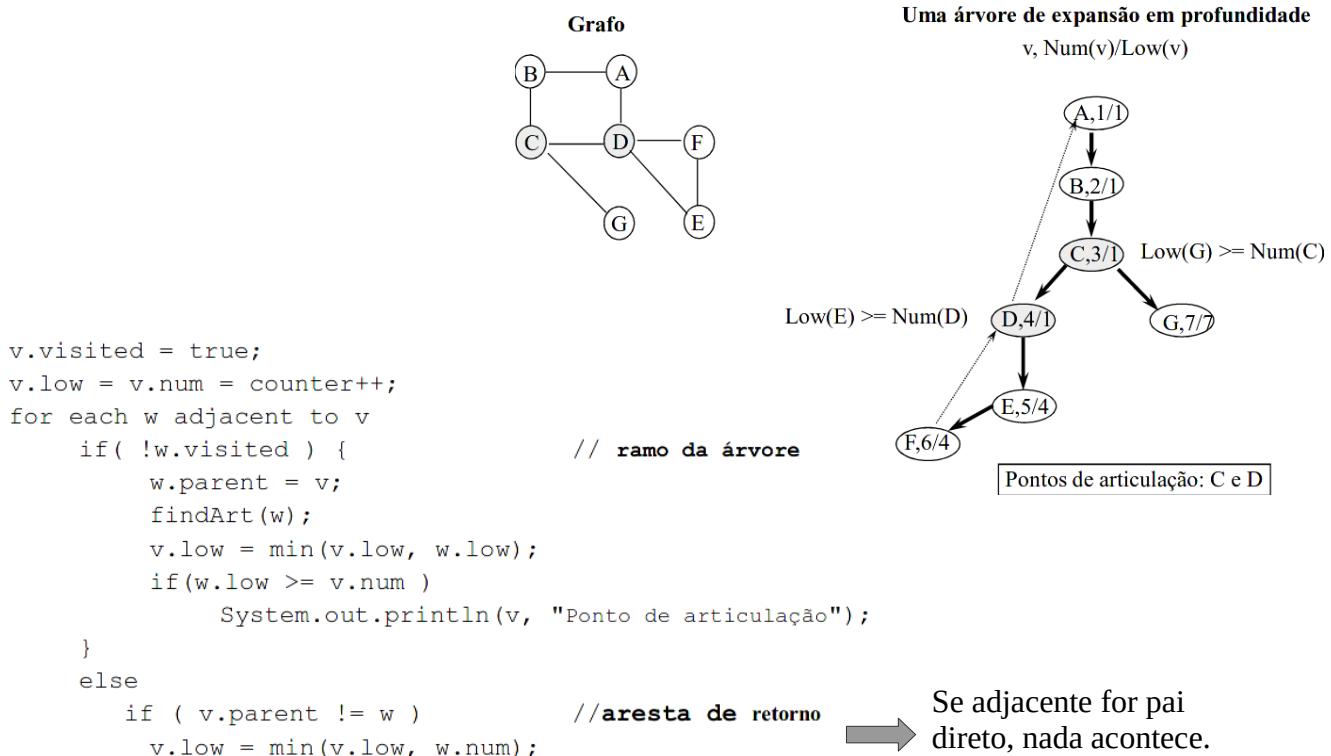
- Um grafo não dirigido é conexo se e só se uma pesquisa em profundidade, a começar em um qualquer vértice, visita todos os vértices do grafo.

### Biconectividade e Pontos de Articulação

- Grafo conexo não dirigido é **biconexo** se não existe nenhum vértice cuja remoção torne o resto do grafo desconexo (**Ponto de Articulação**).
- **Algoritmo de deteção 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** (em qualquer ponto da árvore de DFS, quão próximo se pode chegar à raiz). Vértice  $v$  é ponto de articulação se tiver um filho  $w$  tal que  $Low(w)\geq Num(v)$  (a partir de um descendente  $w$

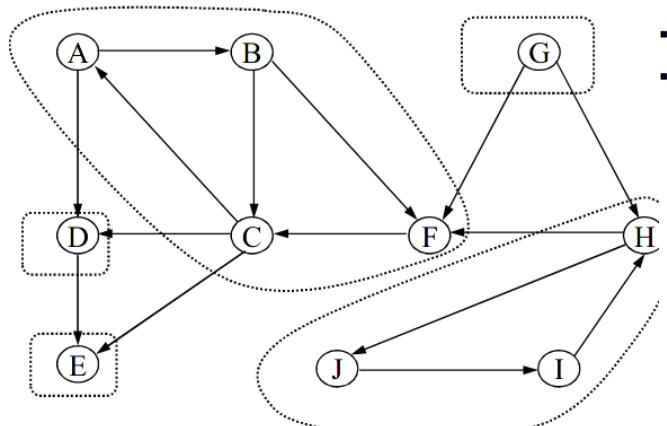
de  $v$  o mais próximo da raiz nunca será menor que o nível onde está localizado o vértice  $v$ . A raiz é ponto de articulação se e só se tiver mais que um filho na árvore.

- $Low(v)$  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  $(v, w)$  (adjacentes) da árvore.
- Na visita de profundidade, inicializa-se  $Low(v)=Num(v)$  antes de visitar adjacentes. Vai-se atualizando o valor depois da visita a cada adjacente.
- $O(|E|+|V|)$

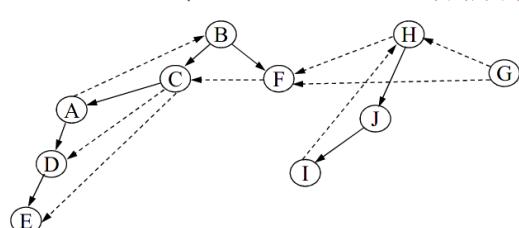


## Grafo dirigido

### Componentes fortemente conexos

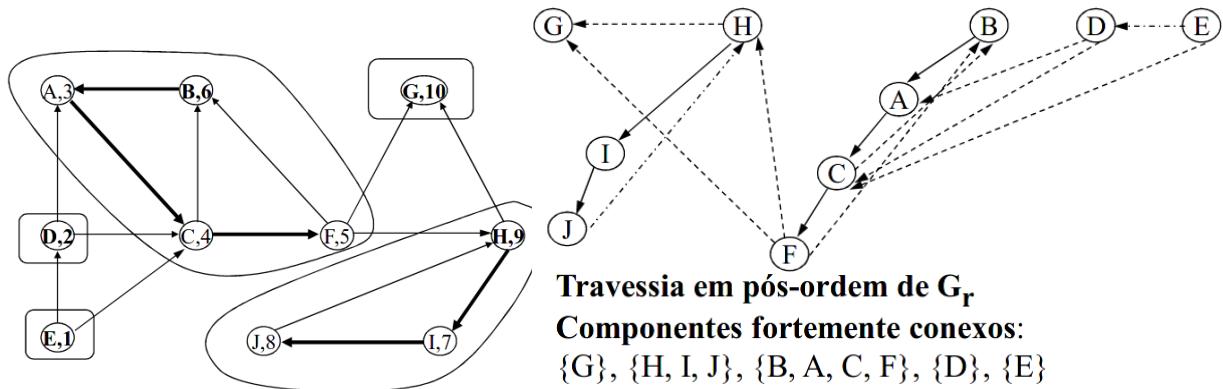


- Pesquisa em profundidade induz uma árvore/floresta de expansão
- Para além das arestas genuínas da árvore, há arestas para vértices já marcados
  - arestas de retorno para um antepassado – (A,B), (I,H)
  - arestas de avanço para um descendente – (C,D), (C,E)
  - arestas cruzadas para um nó não relacionado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



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

### Inversão das arestas e nova visita



$G_r$ : obtido de  $G$  por inversão de todas as arestas

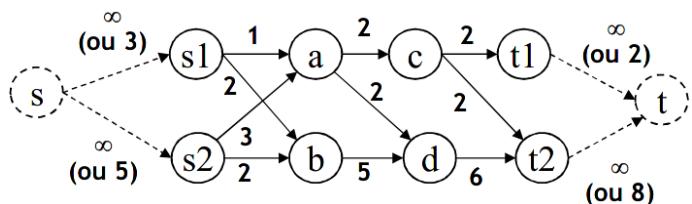
**Numeração:** da travessia de  $G$  em pós-ordem

## Fluxo máximo em Redes de Transporte

- Modelar fluxos conservativos entre dois pontos através de canais com capacidade limitada.
- $s$  : fonte (produtor);  $t$ : poço (consumidor); fluxo não pode ultrapassar a capacidade da aresta; soma dos fluxos de entrada num vértice intermédio é igual à soma dos fluxos de saída (conservativo).
- Por vezes as arestas têm custos associados (custo de transportar uma unidade de fluxo).

### Redes com múltiplas fontes e poços

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



## Problema do fluxo máximo

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

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 :

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

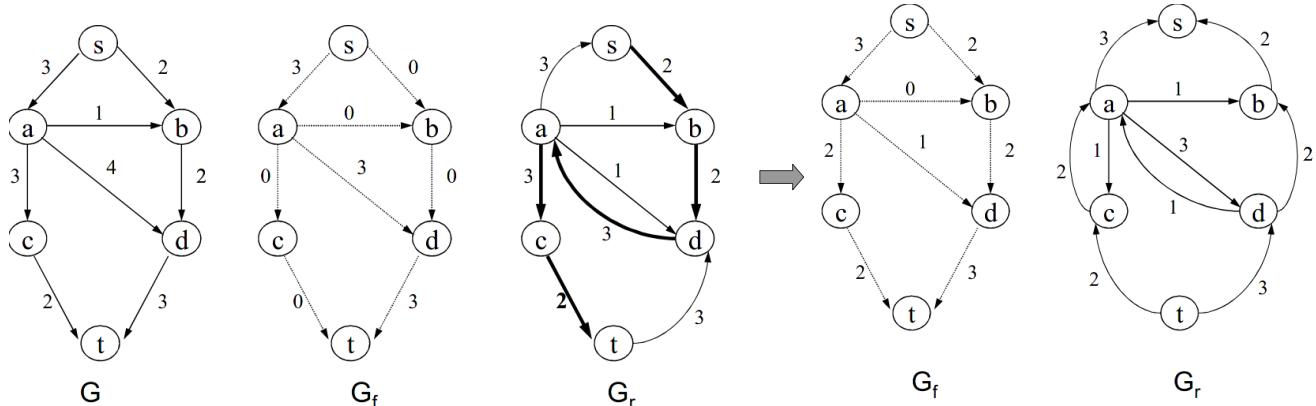
$$\sum_j f_{ij} = \sum_j f_{ji}, \forall_{i \neq s,t}$$

Objectivo :

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

## Ford-Fulkerson

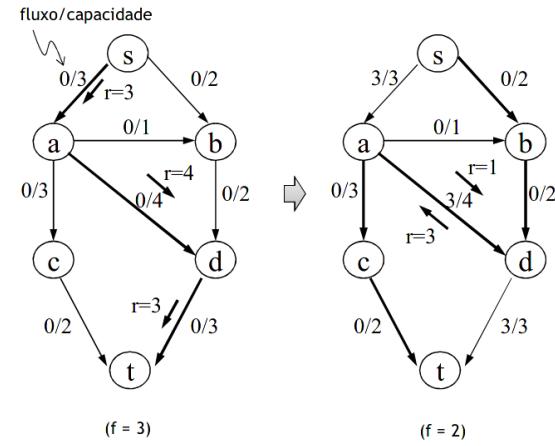
- Estruturas de dados:  $G$  – grafo base de capacidades  $c(v, w)$ ;  $G_f$  - grafo de fluxos  $f(v, w)$ , inicialmente com fluxos iguais a 0, que no fim terá o fluxo máximo;  $G_r$  - grafo de resíduos (auxiliar), em que para cada arco  $(v, w)$  em  $G$  com  $c(v, w) > f(v, w)$ , cria-se um arco 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  $f(v, w)$ .
- 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  $G_r$ .



- Se as capacidades forem números racionais, o algoritmo termina com 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  $O(F \cdot |E|)$

## Edmonds-Karp

- Em cada iteração de Ford-Fulkerson escolhe-se um caminho de aumento de comprimento mínimo. Esse caminho pode ser encontrado em tempo  $O(|E|)$  através de pesquisa em largura. O número máximo de aumentos é  $|E| \cdot |V|$  logo o tempo de execução é  $O(|V| \cdot |E|^2)$
- Uma implementação poderia passar por calcular o grafo de resíduos “on the fly”: Areias percorridas no sentido normal têm resíduo = capacidade – fluxo; Areias percorridas no sentido inverso têm resíduo = fluxo.



```

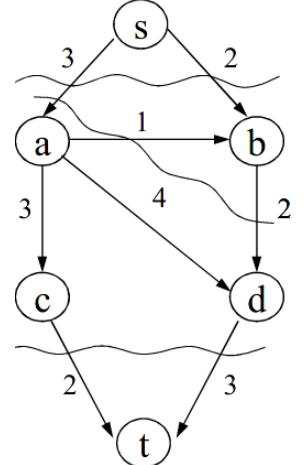
while  $Q \neq \emptyset \wedge \neg \text{visited}(t)$  do
     $v \leftarrow \text{DEQUEUE}(Q)$ 
    for  $e \in \text{outgoing}(v)$  do // direct residual edges
         $\text{TestAndVisit}(Q, e, \text{dest}(e), \text{capacity}(e) - \text{flow}(e))$ 
    for  $e \in \text{incoming}(v)$  do // reverse residual edges
         $\text{TestAndVisit}(Q, e, \text{orig}(e), \text{flow}(e))$ 
    return  $\text{visited}(t)$ 

TestAndVisit( $Q, e, w, \text{residual}$ ):
    . if  $\neg \text{visited}(w) \wedge \text{residual} > 0$  then
    .    $\text{visited}(w) \leftarrow \text{true}$ 
    .    $\text{path}(w) \leftarrow e$  // previous edge in shortest path
    .    $\text{ENQUEUE}(Q, w)$ 

     $e \leftarrow \text{path}(v)$ 
    if  $\text{dest}(e) = v$  then // direct residual edge
         $f \leftarrow \min(f, \text{capacity}(e) - \text{flow}(e))$ 
         $v \leftarrow \text{orig}(e)$ 
    else // reverse residual edge
         $f \leftarrow \min(f, \text{flow}(e))$ 
         $v \leftarrow \text{dest}(e)$ 
     $e \leftarrow \text{path}(v)$ 
    if  $\text{dest}(e) = v$  then // direct residual edge
         $\text{flow}(e) \leftarrow \text{flow}(e) + f$ 
         $v \leftarrow \text{orig}(e)$ 
    else // reverse residual edge
         $\text{flow}(e) \leftarrow \text{flow}(e) - f$ 
         $v \leftarrow \text{dest}(e)$ 
  
```

## 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. Ao identificar este, é obtido o valor do fluxo máximo sem necessitar a utilização de Edmonds-Karp (que é fácil para um grafo simples).



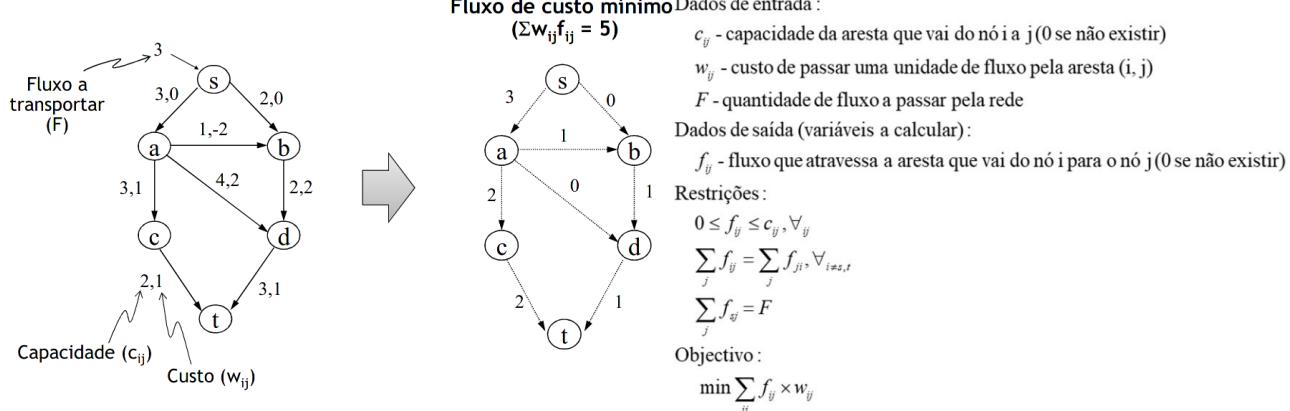
## Algoritmo de Dinic

- Evita trabalho repetido a encontrar caminhos de aumento sucessivos de igual comprimento mínimo.
- Inicializar 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  $\text{nivel}(t) = \infty$ , terminar. “Esconder” as arestas  $(u, v)$  de  $G_r$  em que  $\text{nivel}(v) \neq \text{nivel}(u) + 1$ , logo não podem fazer parte de um caminho mais curto de  $s$  para  $t$  em  $G_r$ . Isto faz com que todos os caminhos entre  $s$  e  $t$  em  $G_r$  têm comprimento mínimo (pesquisa em profundidade).

- Enquanto existirem caminhos de aumento em  $G_r$  (ignorando as arestas escondidas), selecionar 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  $nivel(t) = |V| - 1$ , terminar; senão saltar para o passo 2 (calcular o nível de cada vértice) para recalcular os níveis (voltando a considerar todas as arestas de  $G_r$ ).
- $O(|V|^2 \cdot |E|)$ , melhoria significativa para grafos densos.
- **Rede unitária:** Capacidades unitárias e todos os vértices exceto  $s$  e  $t$  têm no máximo uma aresta a entrar ou uma aresta a sair. Nesse caso o algoritmo de Dinic tem complexidade  $O(|V^{1/2}| \cdot |E|)$ .

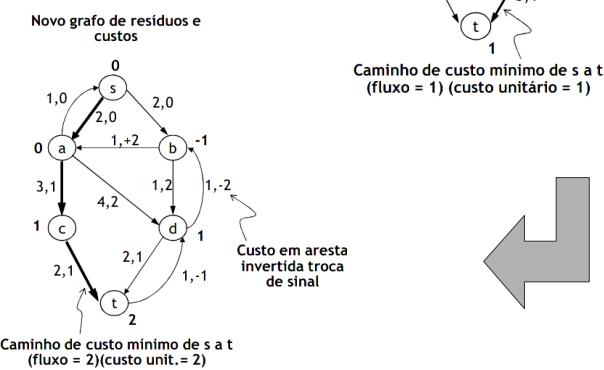
## Fluxo de Custo mínimo em Redes de Transporte

- **Problema:** O objetivo é transportar uma certa quantidade  $F$  de fluxo 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).



## 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). Para-se quando se atinge o fluxo pretendido ou quando não há mais caminhos de aumento (neste caso dá um fluxo máximo de custo mínimo). Dá a solução ótima.
- Restrição: aplicável só a redes sem ciclos de custo negativo. Senão usa-se o método mais genérico (cancelamento de ciclos negativos).

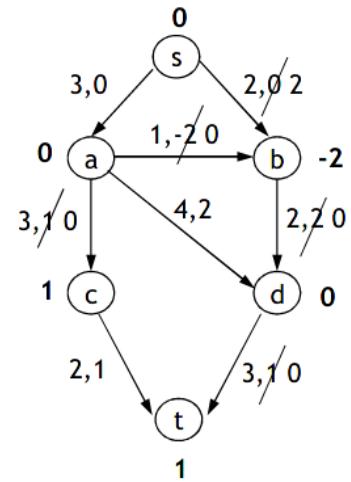


## Melhoramento

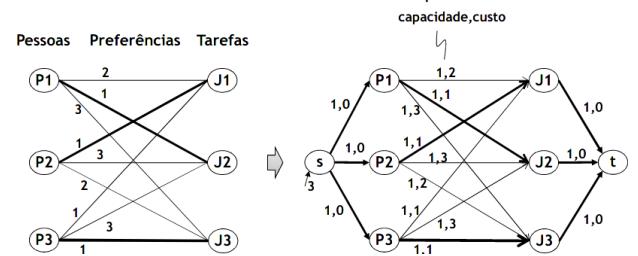
- Dificuldade na abordagem anterior: arestas de custo negativo no grafo de resíduos, o que obriga a usar algoritmo menos eficiente na procura do caminho de custo mínimo (Bellman-Ford).
- Solução:** converte-se o grafo de resíduos num equivalente (para efeito de encontrar caminho de custo mínimo) sem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

## Conversão do grafo de resíduos

- No grafo de resíduos inicial, determinar a “distância” mínima de  $s$  a todos os vértices ( $d(v)$ , que também é chamado neste contexto de “potencial do nó  $v$ ”). Se existirem arestas (mas não ciclos) de peso negativo no grafo de resíduos inicial, usa-se o algoritmo de Bellman-Ford  $O(|V| \cdot |E|)$ .
- 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  $d(t)$ .
- Os caminhos de custo mínimo de  $s$  para  $t$  têm custo reduzido 0 e custo “real” (antes redução)  $d(t)$ . Como os caminhos de custo mínimo percorrem apenas arestas de custo 0, podem ser encontradas com uma pesquisa DFS em tempo linear, conceptualmente, eliminam-se arestas de custo  $> 0$ .
- Ao aplicar o caminho de aumento, o custo das arestas são invertidos (multiplicado por 1). Como  $-1 \times 0 = 0$ , evita-se assim a introdução de arestas de custo negativo.
- Quando não houverem mais caminhos de aumento de custo 0, volta-se a efetuar uma “redução” dos custos no grafo de resíduos (que agora pode ser feito com o algoritmo de Dijkstra pois não há nenhuma aresta negativa  $O(|E| \cdot \log |V|)$ ). Depois, repete-se o processo, procurando pelo caminho com arestas de custo 0.
- Como o número máximo de iterações é  $F$ , tempo total fica  $O(F \cdot |E| \cdot \log |V|)$ .



• 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

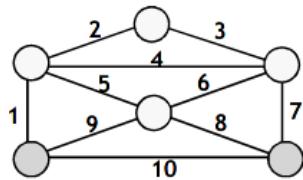


## Circuitos de Euler

- Caminho de Euler:** caminho que visita cada aresta exatamente uma vez.
- Círculo de Euler:** caminho de Euler que começa e acaba no mesmo vértice.

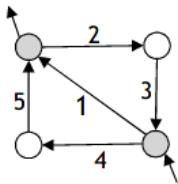
## Condições necessárias e suficientes

### Caminho de Euler



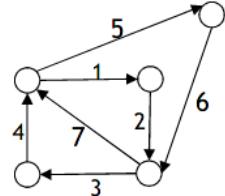
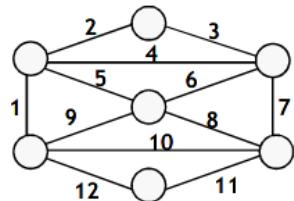
- Um **grafo não dirigido** contém um **círculo de Euler** se e só se é conexo e cada vértice tem grau, ou seja número de arestas incidentes) par.
- Um **grafo não dirigido** contém um **caminho de Euler** se e só se é conexo e todos menos dois vértices têm grau par (estes dois vértices serão os vértices de início e fim do caminho).

### Com caminho de Euler



- Um **grafo dirigido** contém um **círculo de Euler** se e só 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 e só 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.

### Círculo de Euler



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

1. Escolher um vértice qualquer e efetuar uma pesquisa em profundidade a partir desse vértice. Ao visitar um 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 círculo, embora não necessariamente de Euler.

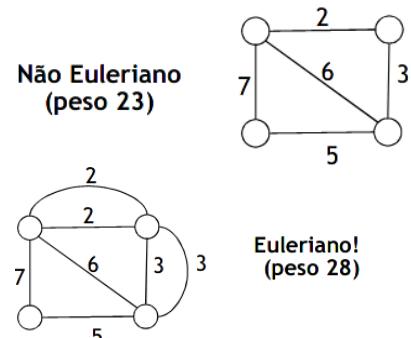
2. Enquanto existirem arestas por visitar: procurar o primeiro vértice no caminho (círculo) obtido até ao momento que possua uma aresta não percorrida; Iniciar uma sub-pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas); Inserir o resultado (círculo) no caminho principal.

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

- 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).

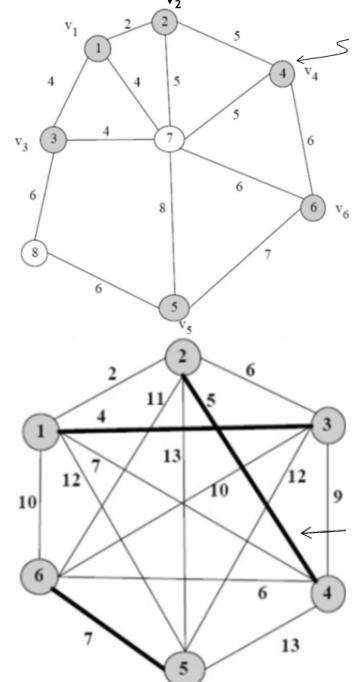
## Problema do carteiro chinês (Chinese postman problem)

- Dado um grafo pesado conexo  $G=(V,E)$ , encontrar um caminho fechado, ou seja, com início e fim no mesmo vértice, de peso mínimo que atravesse cada aresta de  $G$  pelo menos uma vez. Um caminho assim chama-se *percurso ótimo do carteiro chinês*. A um caminho fechado (não necessariamente de peso mínimo) que atravesse cada aresta pelo menos uma vez chama-se *percurso do carteiro*.
- 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).
- Se o grafo  $G$  for Euleriano (um grafo que tem um circuito 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 de forma a conseguir um grafo Euleriano com peso total mínimo.



## Método para grafos não dirigidos

- Passo 1: Achar todos os vértices de grau ímpar em  $G$ . seja  $k$  o número (par) destes vértices. Se  $k = 0$ , fazer  $G^* = G$  e saltar para o passo 6.
- Passo 2: Achar os caminhos mais curtos e distâncias mínimas entre todos os pares de vértices de grau ímpar em  $G$  (p.ex entre  $v_1$  e  $v_2$  é 2, entre  $v_1$  e  $v_3$  é 4, entre  $v_1$  e  $v_5$  é 12, entre  $v_2$  e  $v_3$  é 6 ( $v_2$  e  $v_1$  já se sabe), etc.).
- 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 as somas das distâncias entre vértices emparelhados.
- 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: Encontrar um circuito de Euler em  $G^*$ . Este circuito é um percurso ótimo do carteiro Chinês



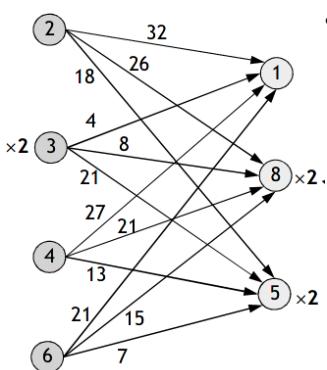
## Passo 4 – Emparelhamento

- 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

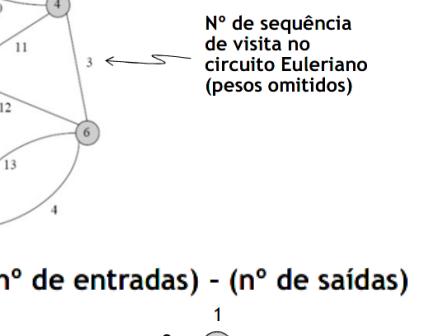
- Um emparelhamento de peso máximo num grafo genérico pode ser encontrado em tempo polinomial (ver referências).

## Método para grafos Dirigidos

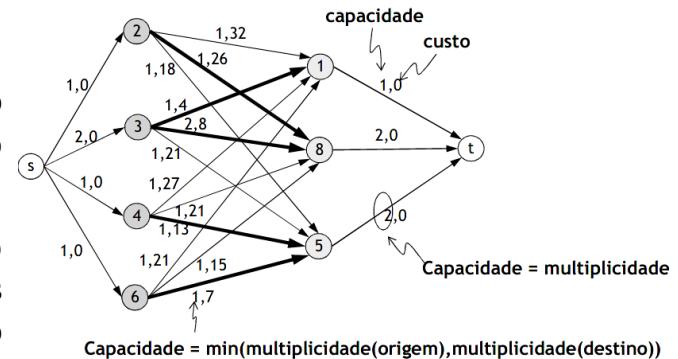
- Passo 1: No grafo  $G$  dado, identificar os vértices com números diferentes de arestas a entrar e a sair.



- Passo 2: Determinar os caminhos mais curtos dos vértices que têm défice de saídas para vértices que têm défice de entradas e representar as distâncias respetivas num grafo bipartido  $G'$ . Vértices são anotados com multiplicidade (número de parelhas em que deve de participar) igual ao défice absoluto.



- Passo 3: Formular problema de emparelhamento ótimo como problema de fluxo máximo de custo mínimo e resolver.
- Passo 4: Obter o grafo Euleriano  $G^*$ , duplicando em  $G$  os caminhos mais curtos entre os vértices emparelhados no passo 3, e obter um circuito Euleriano.

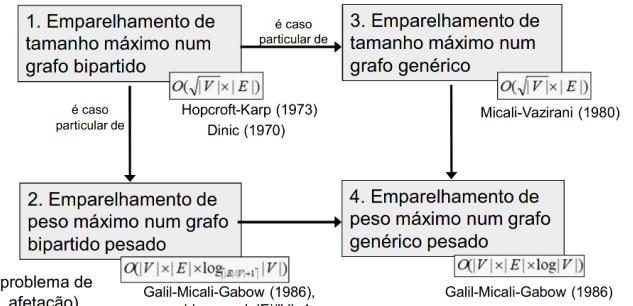


## Problemas de Emparelhamento (matching) e Casamentos Estáveis (stable marriage)

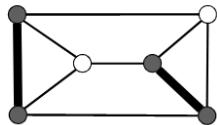
- Seja o grafo não dirigido  $G = (V, E)$
- Formalmente, um **emparelhamento** (*matching*)  $M$  em  $G$  é um conjunto de arestas que não contém mais do que uma aresta incidente no mesmo vértices. Também é chamado conjunto de arestas independentes.
- Emparelhamento maximal:** não pode ser aumentado.

- **Emparelhamento máximo:** tem tamanho máximo; Não é necessariamente único, é necessariamente maximal e o número  $v(G)$  é o tamanho do emparelhamento máximo

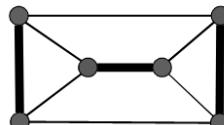
- **Emparelhamento perfeito:** inclui todos os vértices.



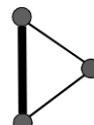
Emparelhamento maximal



Emparelhamento máximo (e perfeito)



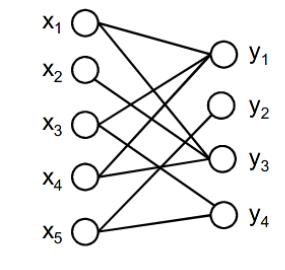
Emparelhamento máximo (mas não perfeito)



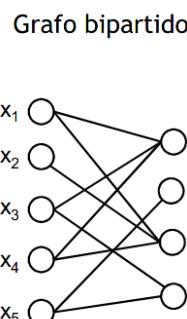
## Redução a problemas em redes de transporte

- Problemas de emparelhamento em grafos bipartidos são redutíveis a problemas em redes de transporte (com capacidades unitárias): Emparelhamento de tamanho máximo  $\rightarrow$  fluxo máximo; Emparelhamento de peso máximo  $\rightarrow$  fluxo de custo mínimo (custo = -peso).
- Grafos genéricos sem ciclos de tamanho ímpar são redutíveis a grafos bipartidos. Basta fazer uma pesquisa em largura, a qual gera uma floresta de pesquisa em largura, 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 complicados.

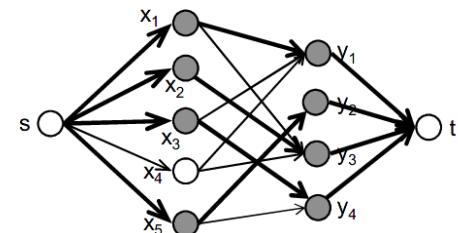
## Emparelhamento de tamanho máximo (bipartido)



e.g. pessoas candidatam-se a empregos



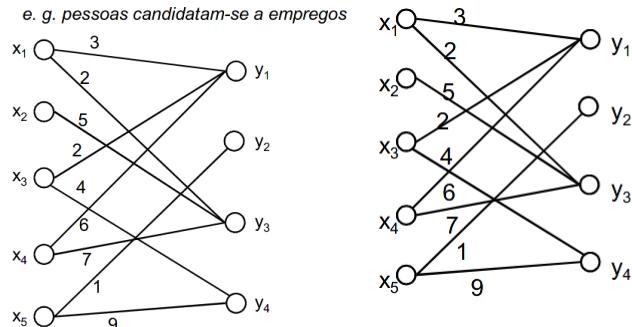
Rede de transporte correspondente (com capacidades unitárias) e fluxo máximo (traço forte)



## Emparelhamento de peso máximo (bipartido)

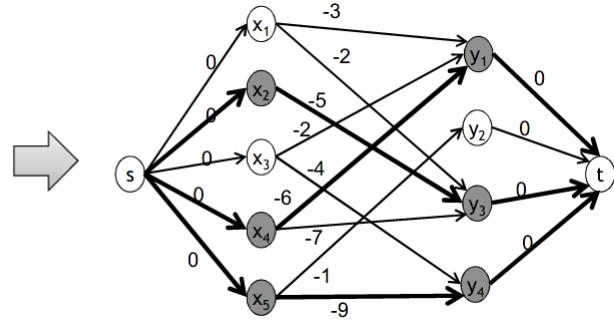
- Capacidades unitárias (logo só se mostram custos e não capacidades).
- Custo do transporte = simétrico do peso do emparelhamento (origina arestas de custo negativo, mas não há ciclos).
- Aplica-se métodos dos caminhos de aumento de custo mínimo; para-se quando o próximo caminho de aumento tem custo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

e.g. pessoas candidatam-se a empregos

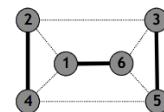
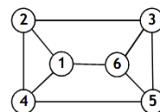


preferência, aptidão, etc.

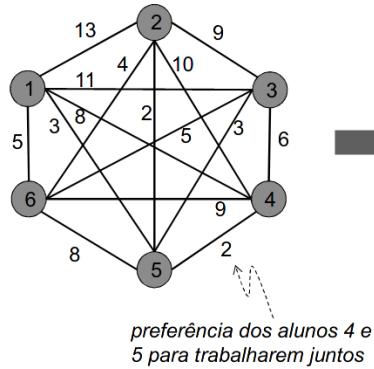
(não ter aresta é o mesmo que ter peso negativo)



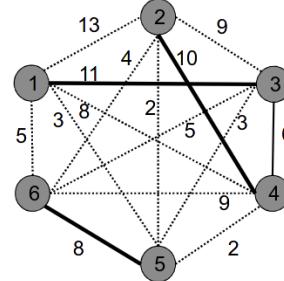
## Emparelhamento de tamanho máximo num grafo genérico



## Emparelhamento de peso máximo num grafo genérico



Neste exemplo o grafo é completo



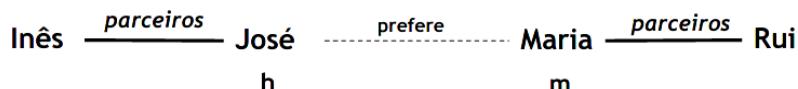
Peso total: 29

(estratégia gananciosa dava 25)

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

## Casamentos Estáveis

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



- O algoritmo garante estabilidade, no entanto, a ordem como as propostas são feitas (se o homem propõe ou a mulher) tem impacto no resultado final.

## ALGORITMO DE GALE-SHAPLEY (1962)

Considerar inicialmente que todas as pessoas estão livres.

Enquanto houver algum homem  $h$  livre fazer:

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

    se  $m$  estiver livre então

        emparelhar  $h$  e  $m$  (ficam noivos)

    senão

        se  $m$  preferir  $h$  ao seu actual noivo  $h'$  então

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

        senão

$m$  rejeita  $h$  e assim  $h$  continua livre.

fim

**Tempo de execução:  $O(n^2)$**

## Lista de preferências incompletas

- Surgiu na colocação de internos em hospitais. O critério de estabilidade das soluções é reformulado:  
Um emparelhamento é instável se e só se existir um candidato  $r$  e um hospital  $h$  tais que
  - $h$  é aceitável para  $r$  e  $r$  é aceitável para  $h$ , e  $r$  não ficou colocado ou prefere  $h$  ao seu atual hospital, e  $h$  ficou com vagas por preencher ou prefere  $r$  a pelo menos um dos candidatos com que ficou.

## ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS)

Considerar inicialmente que todos os internos estão livres.

Considerar também que todas as vagas nos hospitais estão livres.

Enquanto existir algum interno  $r$  livre cuja lista de preferências é não vazia

    seja  $h$  o primeiro hospital na lista de  $r$ ;

    se  $h$  não tiver vagas

        seja  $r'$  o pior interno colocado provisoriamente em  $h$ ;

$r'$  fica livre (passa a não estar colocado);

        colocar provisoriamente  $r$  em  $h$ ;

    se  $h$  ficar sem vagas então

        seja  $s$  o pior dos colocados provisoriamente em  $h$ ;

        para cada sucessor  $s'$  de  $s$  na lista de  $h$

            remover  $s'$  e  $h$  das respectivas listas

fim

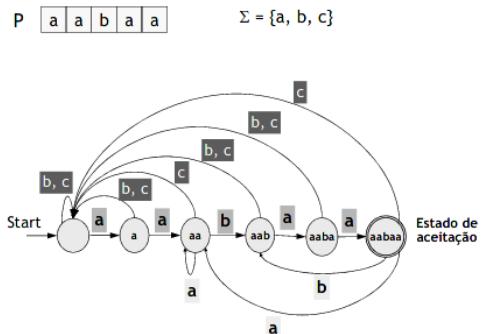
Orientado por internos, logo o resultado será mais favorável aos internos.

**Tempo de execução:  $O(n^o \text{ internos} \times n^o \text{ hospitais})$**

# Pesquisa em Strings

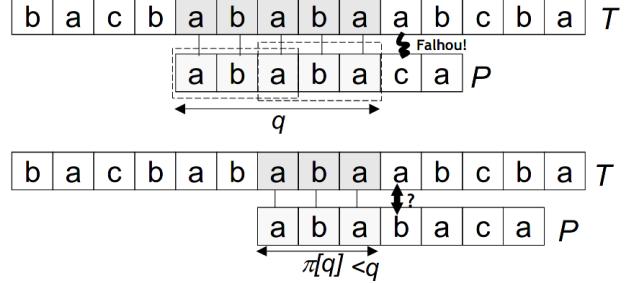
## Pesquisa exata (String Matching)

- Problema:** Encontrar todas as ocorrências de um padrão  $P$  num texto  $T$ .
- $P$  e  $T$  são cadeias de caracteres. Ocorrências são definidas pela deslocação em relação ao início do texto. Ocorrências podem ser sobrepostas.
- Algoritmo naïve:** Para cada deslocamento possível, compara desde o início do padrão. Ineficiente se o padrão for comprido  $O(|P| \cdot |T|)$
- Algoritmo baseado num 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. No entanto, 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 (KMP):** 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 um total de  $O(|T| + |P|)$ .

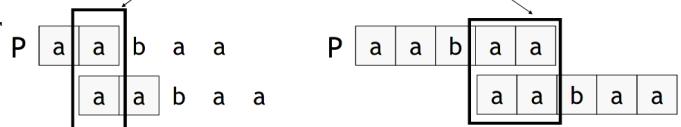


## Knuth-Morris-Pratt

- Desloca-se o padrão para a direita de uma forma que permite continuar a comparação na mesma posição do texto, o que evita comparações inúteis. Deslocamento é determinado por uma função  $\pi[q]$  calculada numa fase de pré-processamento do padrão.
- Compara-se o padrão com deslocações do mesmo, para determinar a função prefixo.
- $\pi[q] = \max \{k : 0 \leq k < q \wedge P[1..k] = P[(q-k+1)..q]\}$  ;
- $q = 1, \dots, |P|$  ;
- $P[i..j]$  - substring entre índices  $i$  e  $j$ ; Índices a começar em 1;
- $\pi[q]$  é o comprimento do maior prefixo de  $P$  que é igual ao sufixo próprio de  $P$  até ao carácter  $q$ .



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



```

std::vector<int> pi = computePrefix(pattern);
int numMatch = 0, matches = 0;

for(char ch : text) {
    while(numMatch > 0 && pattern[numMatch] != ch) {
        numMatch = pi[numMatch - 1]; // character does not match
    }
    if(pattern[numMatch] == ch) {
        numMatch++; // character matches
    }
    if(numMatch == pattern.size()) {
        matches++;
        numMatch = pi[numMatch - 1];
    }
}

```

KMP MATCHER

```

pi[0] = 0;
int j = 0;

for(size_t i = 1; i < patternSize; i++) {
    while(j > 0 && pattern[j] != pattern[i]) {
        j = pi[j-1];
    }
    if(pattern[j] == pattern[i]) j++;
    pi[i] = j;
}

```

COMPUTE-PREFIX.

Índices a começar com 0.

## Pesquisa Aproximada

- **Input:** Uma string T e um padrão P. Um custo k.
- **Problema:** É possível transformar T em P usando no máximo k inserções, remoções ou substituições? (ou: qual é o grau de semelhança entre P e T?).
- **Distância de edição entre P (pattern) e T (text)** é o menor número de alterações necessárias para transformar T em P, em que as alterações podem ser: substituir um carácter por outro, inserir um carácter ou eliminar um carácter.

## 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:  $D[0, j] = j$ ,  $D[i, 0] = i$  ← condições na fronteira da matriz.
- 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 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  $T[j]$ )

		T												
		b	c	d	e	f	f	g	h	i	x	k	l	
P		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	2	2	3	4	5	6	7	
h	8	7	6	5	4	3	3	3	2	3	4	5	6	
i	9	8	7	6	5	4	4	4	3	2	3	4	5	
j	10	9	8	7	6	5	5	4	3	3	4	5		
k	11	10	9	8	7	6	6	5	4	4	3	4		
l	12	11	10	9	8	7	7	6	5	5	4	3		

### Pseudo-código

```

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|]
}

```

Tempo e espaço:  $O(|P| \cdot |T|)$

# Compressão de texto (compression)

- Codificador na fonte e descodificador no destino. (Algoritmos de compressão e algoritmos de descompressão).

## Representação de caracteres

- ASCII: American Standard Code for Information Interchange
- 8 bits para representar um carácter, logo consegue representar 256 caracteres diferentes (Unicode 16 bits, ISO 32 bits).

## 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.

## 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  
AABBBBBBBBAMMKKKKKKKKKM → AA\*B8AMM\*K9M

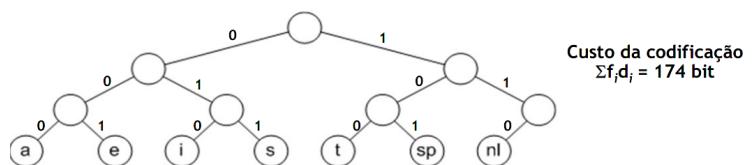
# Algoritmo de Huffman

## Codificação constante

- Código de tamanho fixo. Se  $|\text{alfabeto}| = C \rightarrow$  código com  $\lceil \log_2(C) \rceil$  bits.

- Representação possível: Árvore binária com caracteres só nas folhas. Se é folha, então encontrou-se o carácter; se o bit corrente do código for 0, visita-se a sub-árvore da esquerda; se o bit corrente do código for 1, visita-se a sub-árvore da direita.

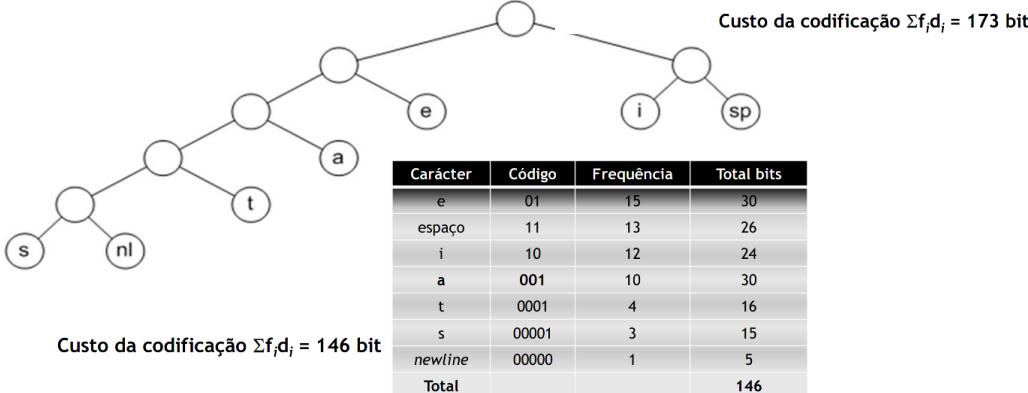
Carácter	Código	Frequência ( $f_i$ )	Total bits
a	000	10	30
e	001	15	45
i	010	12	36
s	011	3	9
t	100	4	12
espaço	101	13	39
newline	110	1	3
<i>Total</i>			



## Codificação variável

Códigos de tamanho variável.

- Codificação óptima...



## Códigos de Huffman

- Código de tamanho variável, caracteres mais frequentes → código mais pequeno.
- Utiliza uma árvore binária com os símbolos só nas folhas. Os símbolos nas folhas permitem descodificação não ambígua (código não prefixo).
- Ao usar uma árvore completa (full tree) todos os nós da árvores (exceto folhas) têm dois descendentes.
- Minimiza o custo da codificação  $\sum f_i \cdot d_i$  onde  $f_i$  é a frequência relativa e  $d_i$  é a profundidade na árvore.
- O algoritmo de Huffman consiste em 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) somando os seus pesos. Repetir o passo anterior até haver uma só árvore. Empates resolvidos aleatoriamente.

# Problemas NP-Completos

## Problemas P

- Considera-se normalmente que um **problema** é **resolúvel eficientemente** se for **resolúvel em tempo polinomial**, ou seja, se houver um **algoritmo de tempo polinomial que o resolva**.
- Um algoritmo é de **tempo polinomial** se o tempo de execução é da ordem de  $O(n^k)$ , no pior caso, em que **n é o tamanho do input** do problema e **k é uma constante independente de n**. (superpolinomial se o tempo não é limitado por nenhum polinómio).
- Algumas funções parecem não ser polinomiais, mas podem ser tratadas como tal:  $O(n \cdot \log n)$  tem delimitação superior  $O(n^2)$ .
- Algumas funções parecem ser polinomiais, mas podem não o ser na verdade:  $O(n^k)$ , se k variar em função de n, tamanho do input.
- A **classe de problemas P** é constituída por todos os **problemas de decisão que podem ser resolvidos em tempo polinomial**.

## 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 otimização (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 só 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?”.

## Problemas NP

- A **classe de problemas NP** (nondeterministic polynomial) é definida por todos os problemas que podem ser verificados por um algoritmos de tempo polinomial.
- Não confundir **resolução** em tempo polinomial (P) com **verificação** em tempo polinomial (NP).

## Problema do Circuito Hamiltoniano

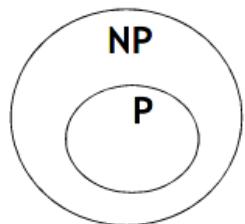
- Problema do circuito Hamiltoniano não dirigido (Undirected Hamiltonian Cycle – UHC): verificar se um grafo não dirigido dado G, é Hamiltoniano, isto é, tem um ciclo (ou circuito) que visita cada vértice exatamente uma vez.

## 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: por exemplo, o problema de determinar se um grafo G tem exatamente 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 correta em tempo polinomial.
- Não se sabe certamente se  $P = NP$  ou  $P \neq NP$

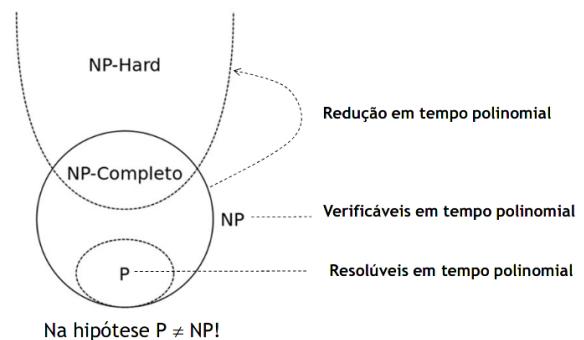


## Problemas NP-completos (NPC)

- 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  $A \in NP$  (i) e qualquer problema  $A' \in NP$  é **redutível em tempo polinomial** a A ( $A' \leq_p A$ ) (ii). 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  $A'$ . Atualmente, para provar que A é NPC, basta encontrar um problema A' NPC já conhecido e provar que A' é redutível a A em tempo polinomial.
- 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 **redutível 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.



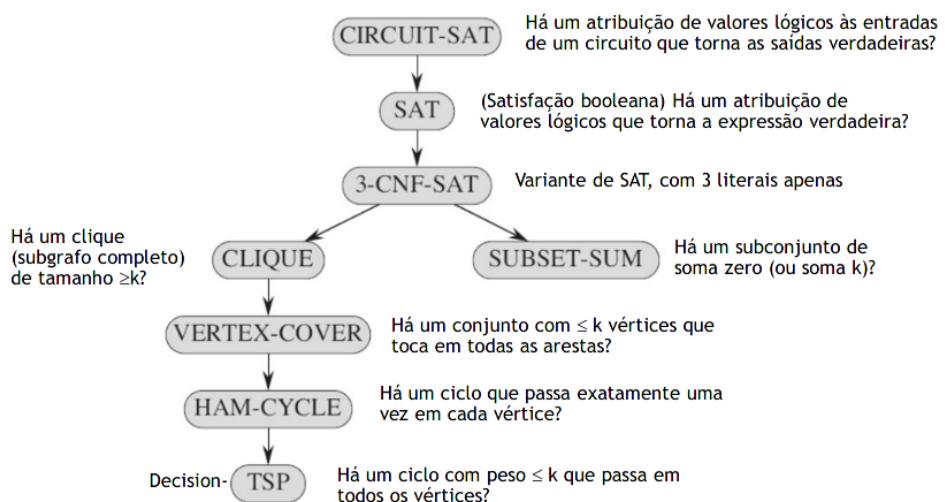
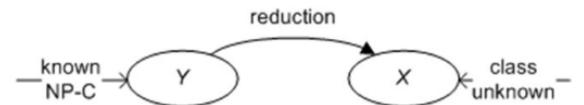
# Classificação de problemas por redução

## Como “provar” que um problema não pertence a P

- Selecionar um problema Y não resolúvel em tempo polinomial ( $Y \notin NP$ ). De acordo com o conhecimento atual, em que se acredita que  $P \neq NP$ .
- Provar que Y é redutível 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.

## Como provar que um problema X pertence a 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).



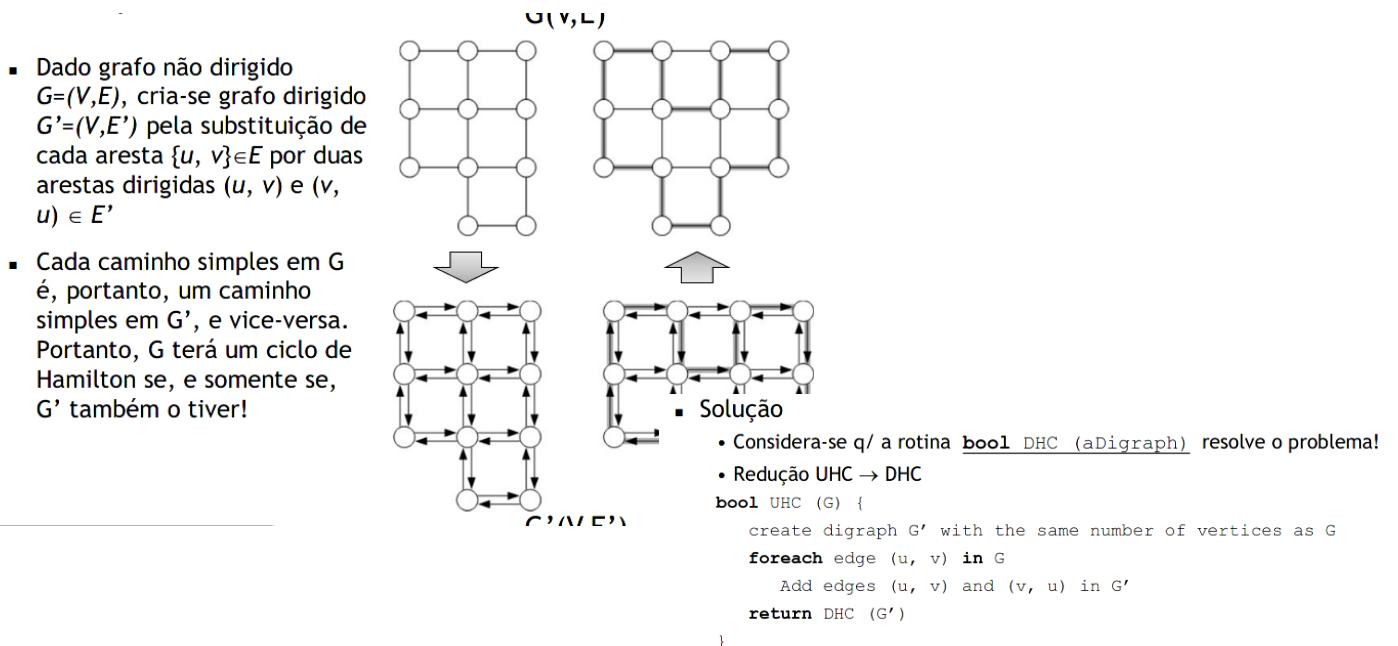
## Redução de problemas NP

- Dados dois problemas, **A** e **B**, diz-se que **A** é polinomialmente redutível a **B** se, dada uma sub-rotina de tempo polinomial para **B**, pode-se utilizá-la para resolver **A** em tempo polinomial. Quando tal-se verifica, expressa-se por  $A \leq_p B$  (**A** redutível a **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:** transitividade

- 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  $\leftarrow$  contradição
- B é NP-completo se  $B \in NP$  e  $A \leq_p B$  para algum problema A, se A é NP-completo.
- Se não se sabe se  $B \in NP$  mas  $A \leq_p B$  para algum problema A, se A é NP-completo, então B é NP-hard.

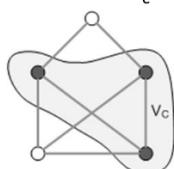
## Directed Hamiltonian Cycle (DHC) é NPC?

- Sabendo que o problema UHC (Undirected Hamiltonian Cycle) é NP-completo, provar que DHC (Directed Hamiltonian Cycle) é também NP-completo
- Um ciclo Hamiltonian candidato é facilmente verificável em tempo polinomial, logo  $DHC \in NP$ .
- O problema UHC é facilmente redutível ao problema DHC em tempo polinomial, logo  $DHC \in NPC$



## 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  $G$ .

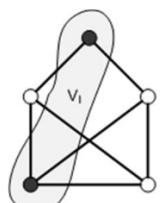
- Problema de decisão (VC):

- O grafo  $G$  tem uma cobertura de vértices de tamanho  $\leq k$ ?

## 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  $G$ .

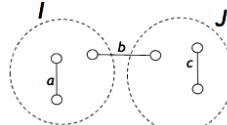


- Problema de decisão (IS):

- O grafo  $G$  tem um conjunto independente de tamanho  $\geq k$ ?

## 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

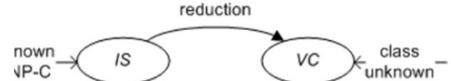


## Vertex Cover é NPC?

- 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  $VC$

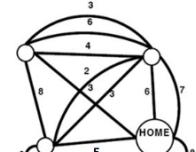
## Redução de IS a VC

- 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  $\geq k$  sse 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 do Caminhada (Jogging (J))

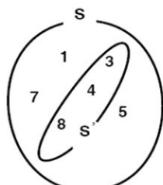
- Considere um grafo não dirigido  $G$ , admitindo arestas paralelas e anéis, com pesos inteiros positivos nas arestas, no qual se distingue um vértice *home*.
- O problema da caminhada (*Jogging (J)*) consiste em verificar se existe um caminho de peso total  $k$ , começando e terminando em *home*, sem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



## Problema da Soma de Subconjuntos (SS)

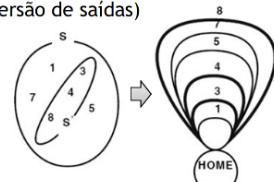
- Dado um conjunto de inteiros positivos,  $S$ , há um subconjunto,  $S'$  em  $S$ , tal que a soma dos elementos de  $S'$  seja  $k$ ?

Ex:  
 $S = \{1, 3, 4, 5, 7, 8\}$   
 Find  $S'$  with sum = 15!



## Problema da Caminhada é NPC?

- Um caminho candidato é facilmente verificável em tempo polinomial, logo  $J \in NP$
- Para provar que  $J \in NP-hard$ , 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  $k$  sse  $G$  tem um caminho de peso total  $k$  sem repetir arestas (conversão de saídas)



## Problema da Marcação de Exames

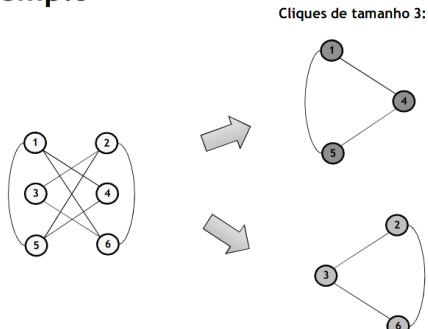
(exame 2016/17)

- 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

## 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

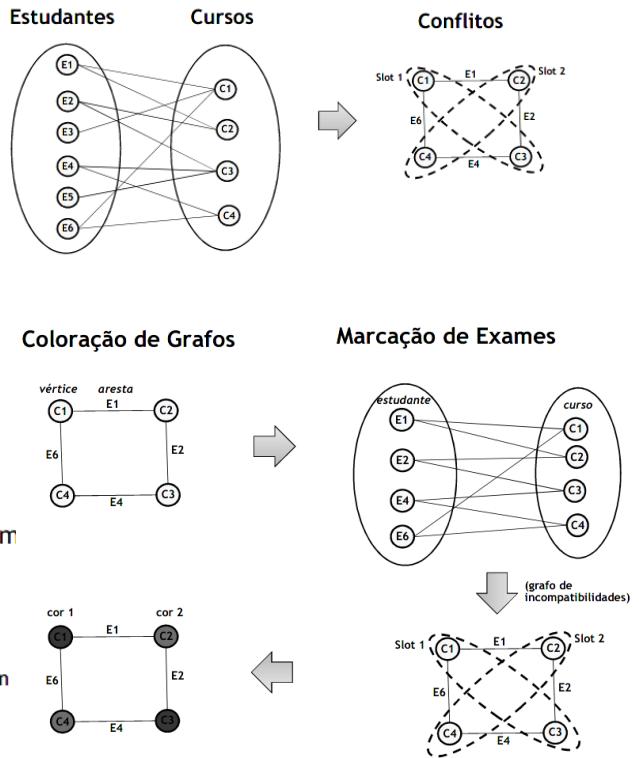
## Exemplo



## 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 redutível 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

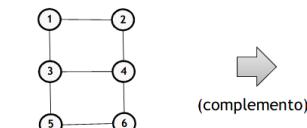
## Exemplo



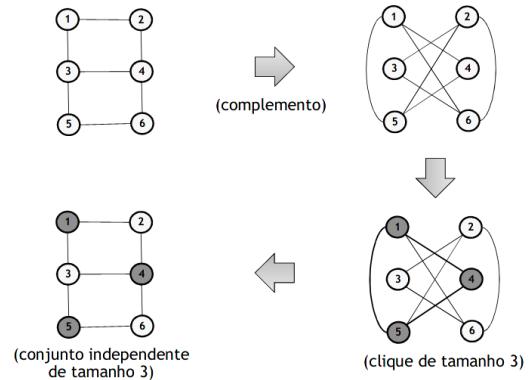
## Problema do Clique (recurso 2016/17)

- 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

### Problema do Conjunto Independente



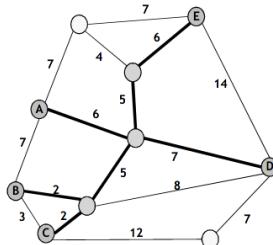
### Problema do Clique



## 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)



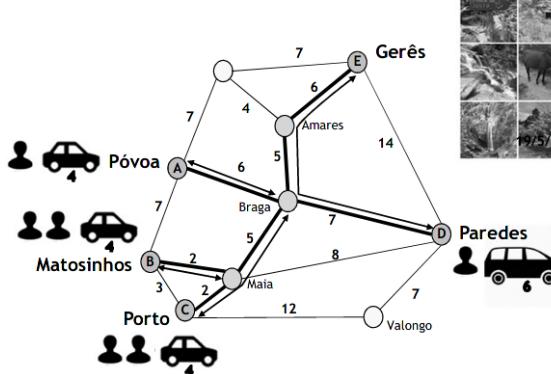
$S = \{A, B, C, D, E\}$

Peso total da árvore: 33

## Problema 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)  $\leq 2k$

# Dynamic Solving

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

## 3º) Derivar fórmulas de cálculo

	i	0	1	2	3	4	5	6=n
Sequência	s <sub>i</sub>	(-∞)	6	3	8	4	7	2
Tamanho L <sub>i</sub>	TL <sub>i</sub>	3	1	2	0	1	0	0
Índ. 1º elem. L <sub>i</sub>	PL <sub>i</sub>	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:  $s_{PL_0}, s_{PL_1}, \dots$  (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)!

- Gera subproblemas do tipo  $L_{>x}S$ , significando “encontrar subsequência crescente mais comprida de  $S$  com valores maiores do que  $x$ ” (podendo-se considerar inicialmente  $x = -\infty$ ).
- Ocorrem subproblemas repetidos, o que sugere a aplicação de programação dinâmica.
- Subproblemas podem ser identificados pelo índice  $i$  de  $x$  na sequência original, ou seja, como  $L_i$ .

Se  $L_i$ 's forem resolvidos iterativamente pela ordem  $L_n, \dots, L_1, L_0$ , evita-se repetição de trabalho (programação dinâmica).  
(No slide anterior, se tivéssemos começado a exploração pelo último elemento, a ordem de iteração seria  $L_0, L_1, \dots, L_n$ .)

Para cada  $L_i$ , em vez de se guardar a solução, basta guardar o tamanho da solução ( $TL_i$ ) e o índice do 1º elemento da solução ( $PL_i$ ), e no final reconstrói-se facilmente a solução ótima completa.

```
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<sup>2</sup>)  
 S(n)=O(n)