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

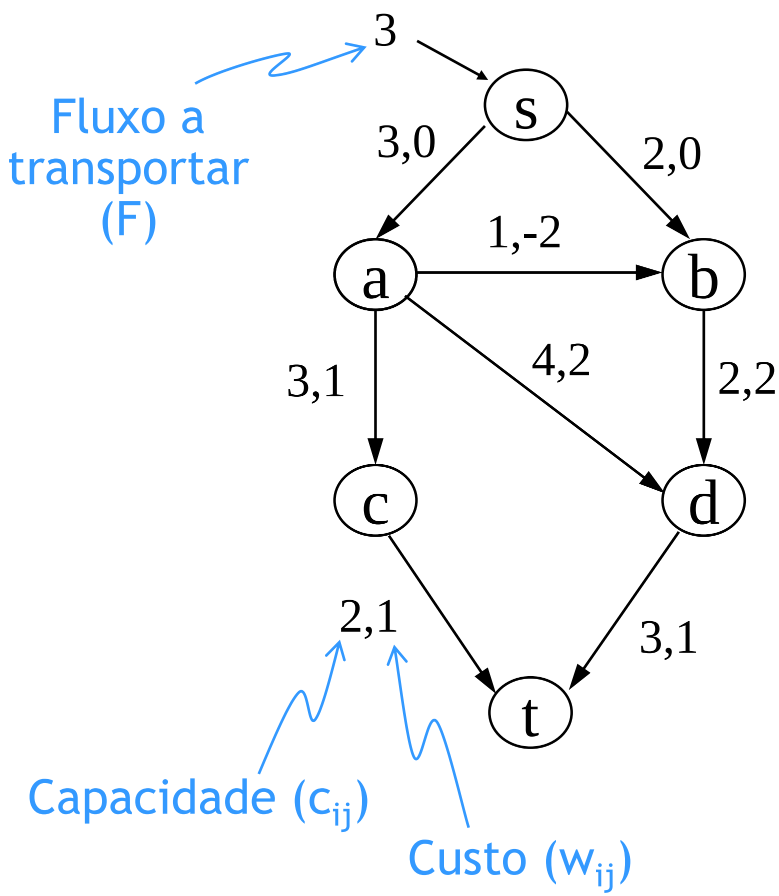
Rosaldo Rossetti

Desenho de Algoritmos, L.EIC

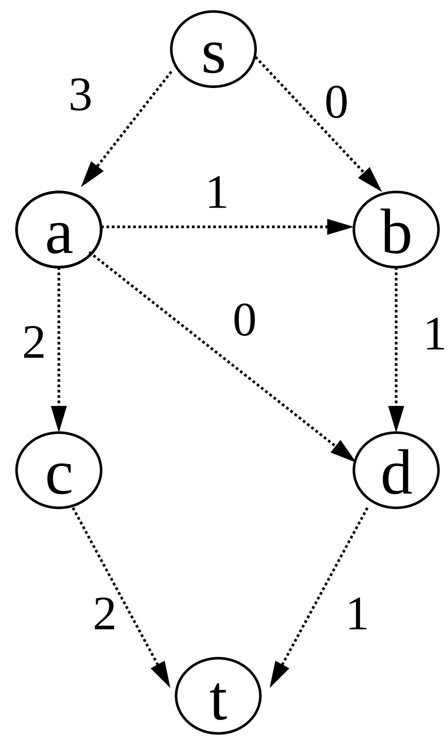
Problema

- O objetivo é transportar uma certa quantidade *F* de fluxo (≤ máximo permitido pela rede) da fonte (*s*) para o poço (*t*), com um custo total mínimo
  - Para além da capacidade, arestas têm associado um custo (w_{ij}, custo de transportar uma unidade de fluxo)
  - Podem existir arestas de custo negativo (útil em problemas de maximização do valor, introduzindo sinal negativo)

Exemplo



Fluxo de custo mínimo
($\Sigma w_{ij} f_{ij}$ = 5)



Formalização

Dados de entrada :

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

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

F - quantidade de fluxo a passar pela rede

Dados de saída (variáveis a calcular) :

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

Restrições :

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

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

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

Objectivo :

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

Algoritmos

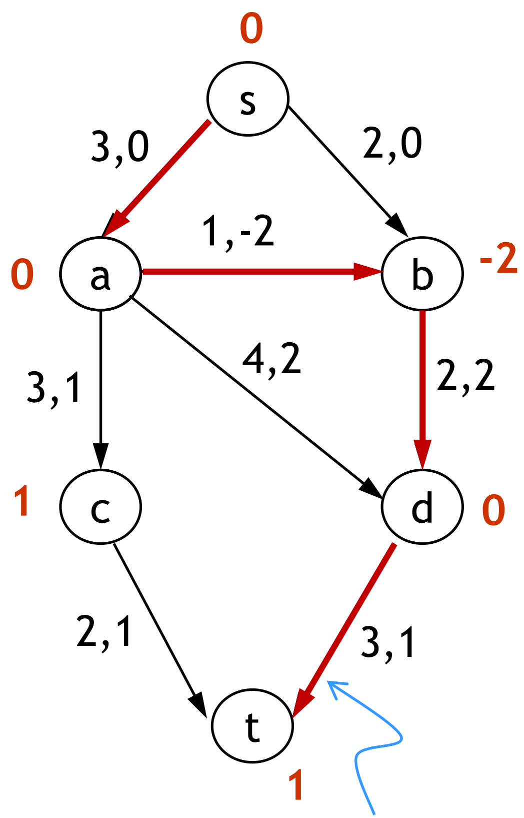
- Há muitos algoritmos propostos na literatura, aplicados à resolução deste problema, incluindo:
  - Cycle cancelling algorithms *(negative cycle optimality)*
  - Successive Shortest Path algorithms *(reduced cost optimality)*
  - Out-of-Kilter algorithms (complimentary slackness)
  - Network Simplex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

Método dos caminhos de aumento mais curtos sucessivos

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

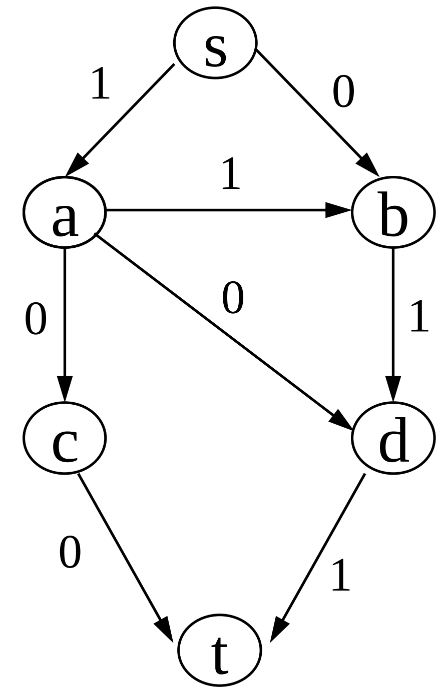
Exemplo (1/2)

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



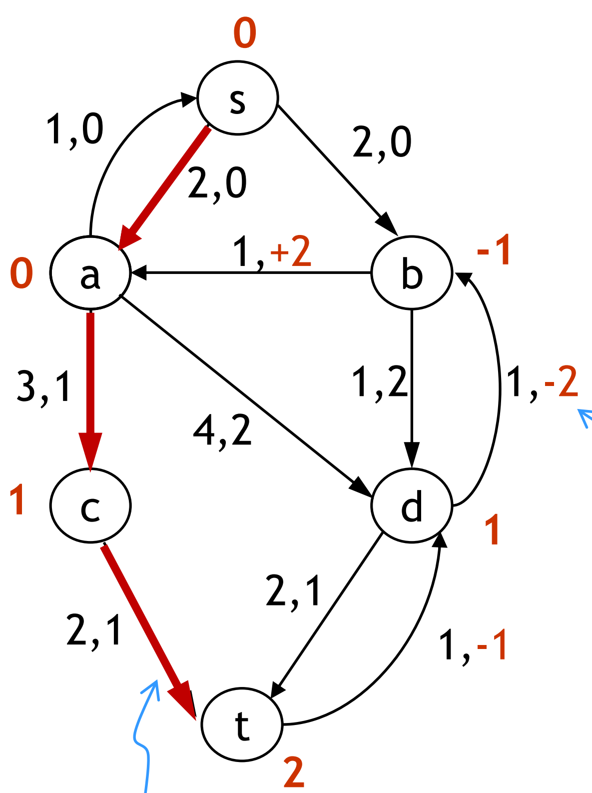
Caminho de custo mínimo de s a t
(fluxo = 1) (custo unitário = 1)

Grafo de fluxos
resultante



Exemplo (2/2)

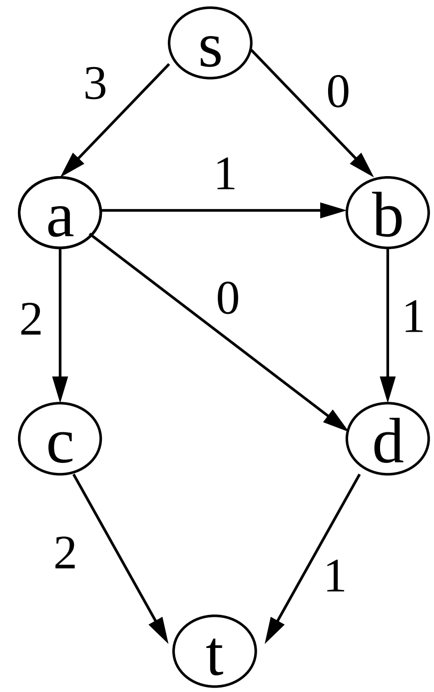
Novo grafo de resíduos e custos



Custo em aresta invertida troca de sinal

Caminho de custo mínimo de s a t (fluxo = 2)(custo unit.= 2)

Grafo de fluxos resultante



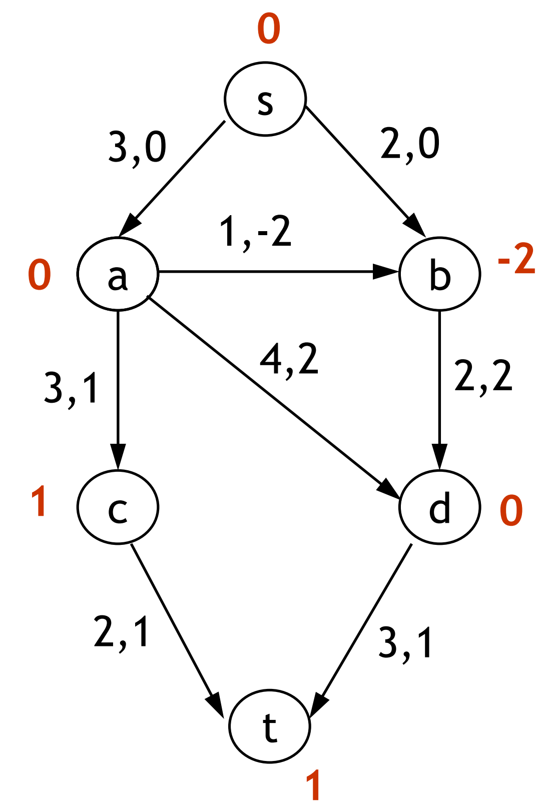
Atingiu-se o fluxo pretendido (3) => Termina (custo total = 5)

Melhoramento

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

Conversão do grafo de resíduos (1/2)

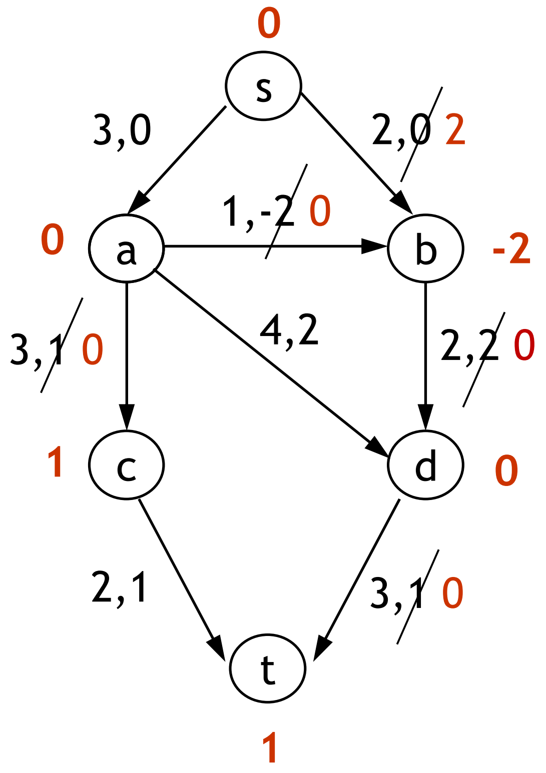
1. No grafo de resíduos inicial, determinar a "distância" mínima de *s* a todos os vértices (*d(v)*)
   - Se existirem arestas (mas não ciclos) de peso negativo no grafo de resíduos inicial, usa-se o algoritmo de Bellman-Ford, de tempo O(|E||V|)
   - *d(v)* também é chamado neste contexto o "potencial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



Conversão do grafo de resíduos (2/2)

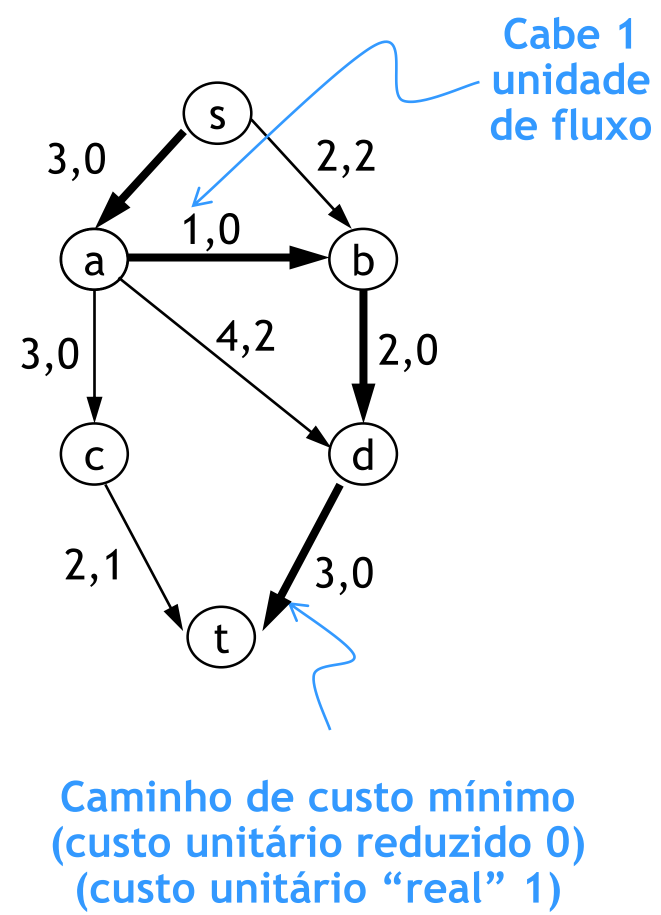
2. Substituir os custos iniciais $w(u,v)$ por custos "reduzidos" $w'(u,v) = w(u,v) + d(u) - d(v)$

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



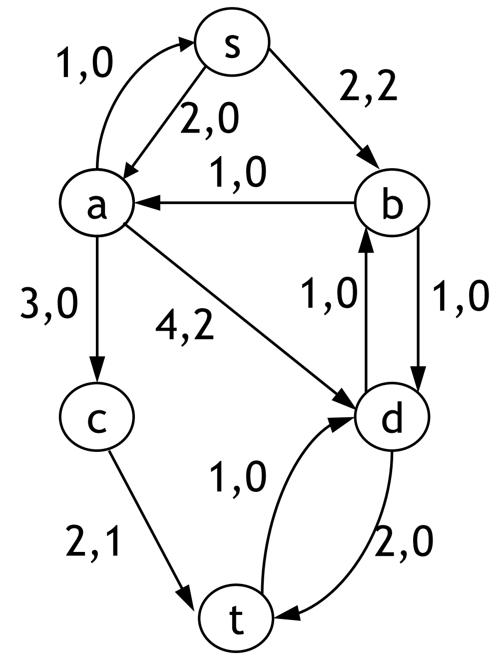
Determinação do próximo caminho de aumento

3. Seleccionar um caminho de custo mínimo de *s* para *t* no grafo de resíduos
   - Os caminhos de custo mínimo de *s* para *t* têm custo reduzido 0 e custo "real" (antes da redução) d(t)
   - Como os caminhos de custo mínimo percorrem apenas arestas de custo 0, podem ser encontrados como uma pesquisa simples (DFS) em tempo linear
     - conceitualmente, eliminam-se arestas de custo > 0



Aplicação do caminho de aumento

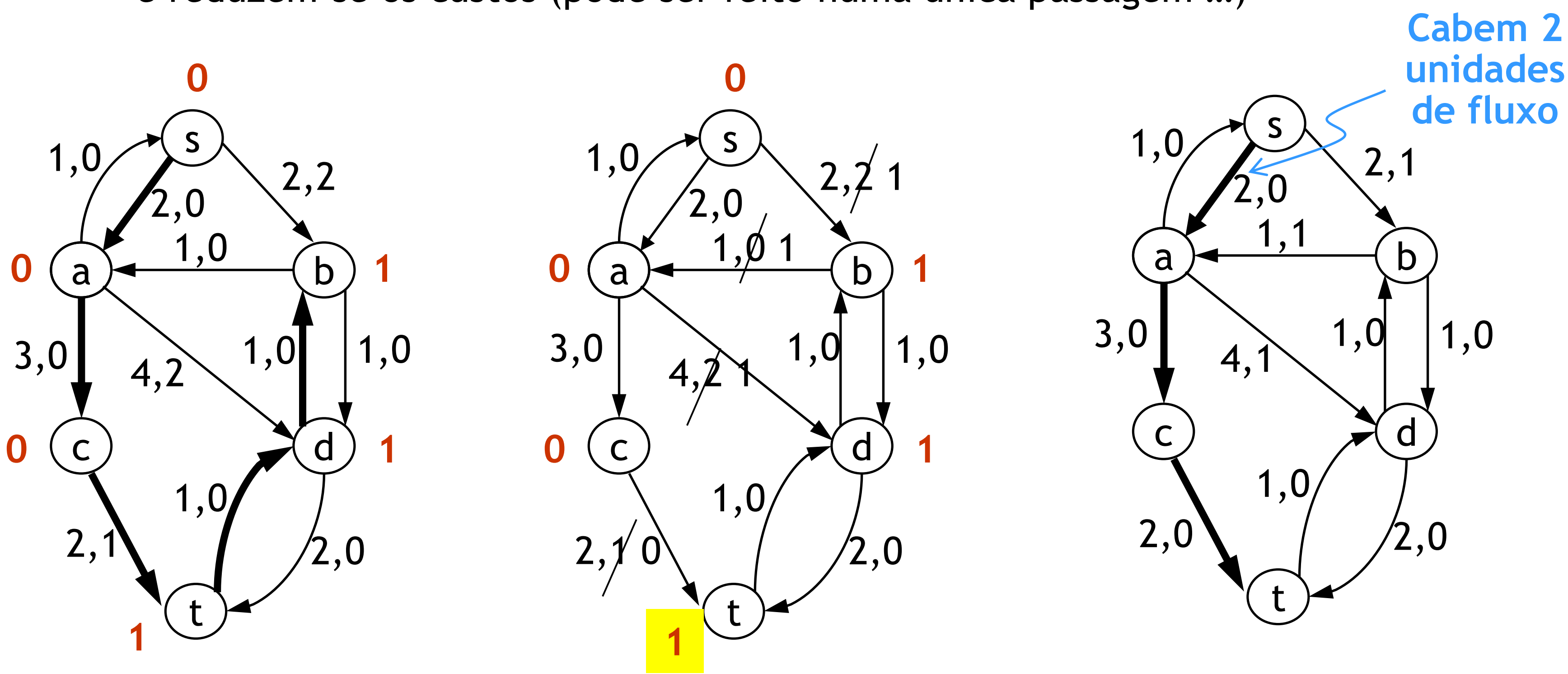
4. Aplicar o caminho de aumento
   - Custo das arestas invertidas no grafo de resíduos é multiplicado por (-1)
   - Só que -1 x 0 = 0 …
   - Evita-se assim a introdução de arestas de custo negativo!



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

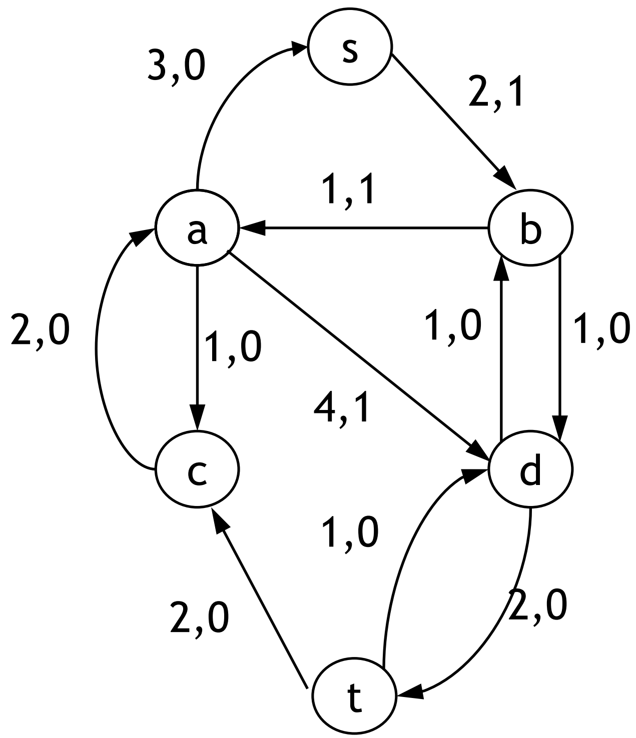
Nova conversão do grafo de resíduos

5. Quando não há mais caminhos de aumento de custo 0, volta-se a efectuar uma "redução" dos custos no grafo de resíduos
   - Isto é, recalculam-se as distâncias mínimas (e.g. pelo algoritmo de Dijkstra!), e reduzem-se os custos (pode ser feito numa única passagem …)



Repetição do processo

6. Aplicar o caminho de aumento de custo 0
   - Actualiza-se o grafo de resíduos



Quantidade atual de fluxo = 3 = pretendido => FIM

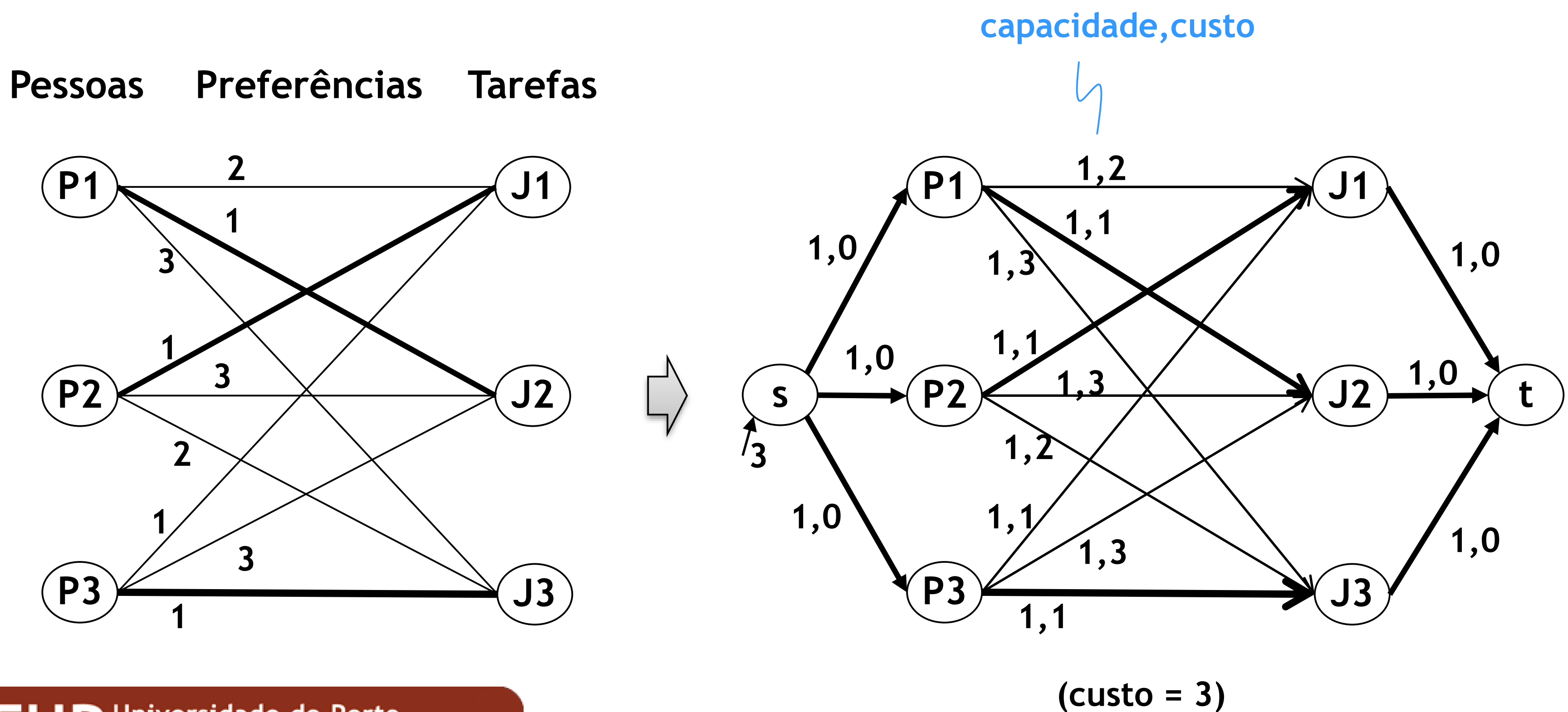
Custo atual do fluxo = 5

Eficiência temporal

- Primeira redução do grafo de resíduos: O(|V| |E|) pelo algoritmo de Bellman-Ford
- Subsequentes reduções do grafo de resíduos e determinação do caminho de aumento de custo mínimo: O(|E| log |V|) pelo algoritmo de Dijkstra
- Se todas as grandezas forem inteiras, o nº máximo de iterações é F, pois em cada iteração o valor do fluxo é incrementado de uma unidade
- Tempo total fica *O*(F |E| log|V|)

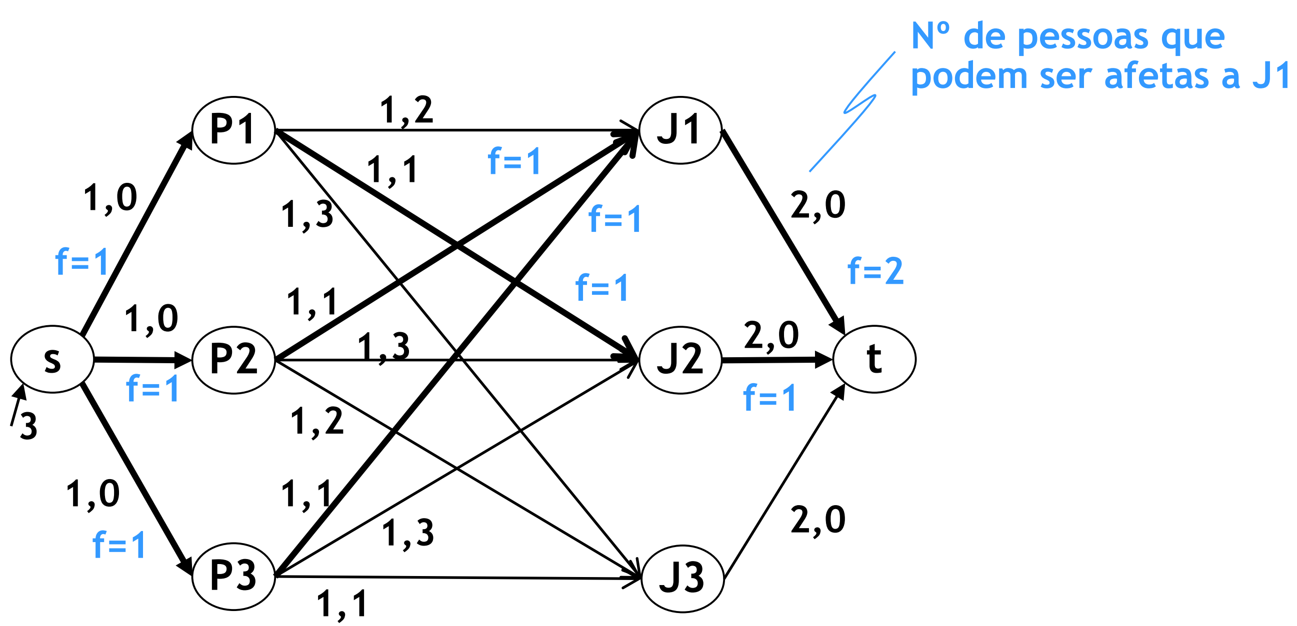
Aplicação a problemas de emparelhamento (1/2)

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



Aplicação a problemas de emparelhamento (2/2)

- E no caso de se poderem afetar várias pessoas à mesma tarefa?
  - Por exemplo, no caso anterior, admitindo 2 pessoas por tarefa



Referências e informação adicional

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

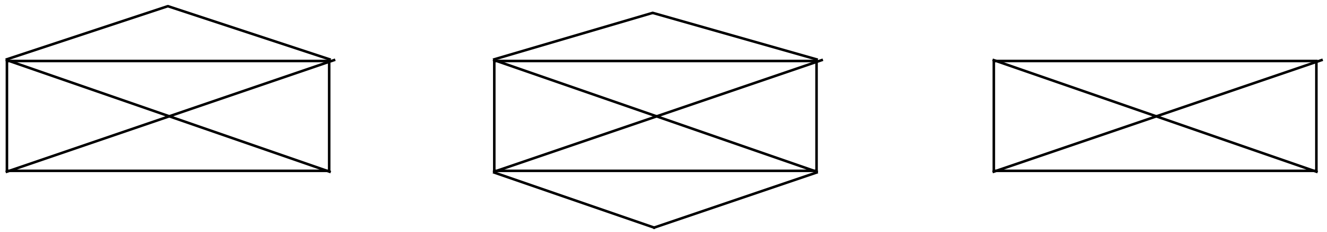
Algoritmos em Grafos: Circuitos de Euler e Problema do Carteiro Chinês

Rosaldo Rossetti

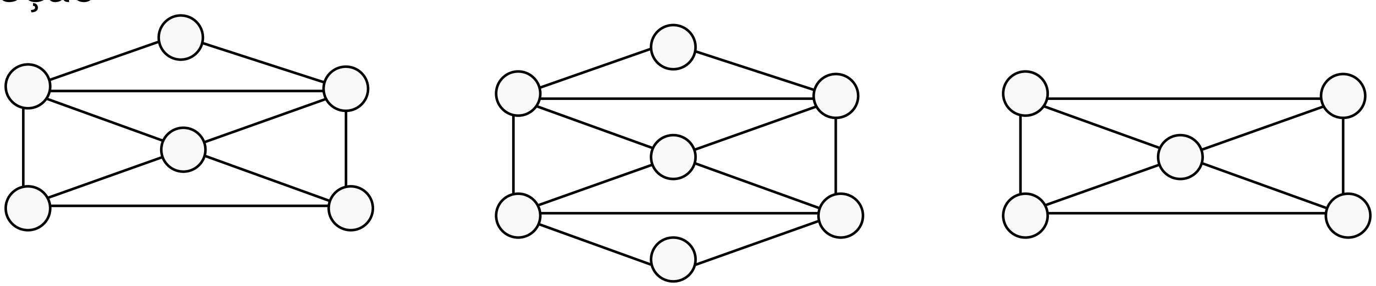
Desenho de Algoritmos, L.EIC

Circuitos de Euler

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



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

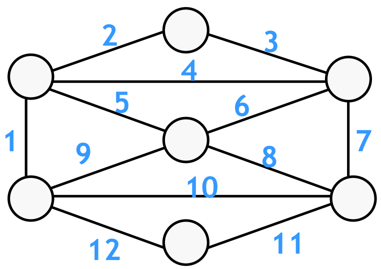


- **Caminho de Euler:** caminho que visita cada aresta exatamente uma vez
- Problema resolvido por **Leonhard Euler** em 1736 e que marca o início da Teoria dos Grafos
- **Circuito de Euler:** caminho de Euler que começa e acaba no mesmo vértice

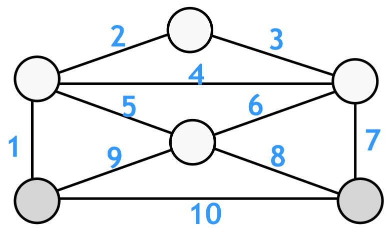
Condições necessárias e suficientes (1/2)

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

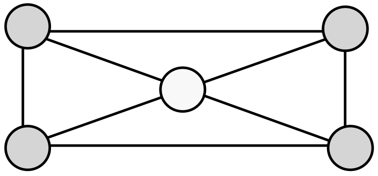
Circuito de Euler



Caminho de Euler



Sem caminho ou circuito de Euler

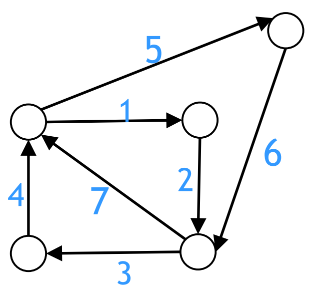


Condições necessárias e suficientes (2/2)

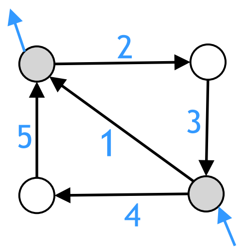
- Um grafo *dirigido* contém um *circuito* de Euler *sse*
  (1) é (fortemente) conexo e
  (2) cada vértice tem o mesmo grau de entrada e de saída.

- Um grafo *dirigido* contém um *caminho* de Euler *sse*
  (1) é (fortemente) conexo e
  (2) todos menos dois vértices têm o mesmo grau de entrada e de saída, e os dois vértices têm graus de entrada e de saída que diferem de 1.

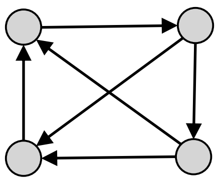
Com circuito de Euler



Com caminho de Euler



Sem circuito ou caminho de Euler



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

1. Escolher um vértice qualquer e efetuar uma pesquisa em profundidade a partir desse vértice
   - Visitar vértice: se tiver arestas incidentes não visitadas, escolher uma dessas arestas, marcá-la como visitada, e visitar vértice adjacente
   - Se o grafo satisfizer as condições necessárias e suficientes, esta pesquisa termina necessariamente no vértice de partida, formando um circuito, embora não necessariamente de Euler

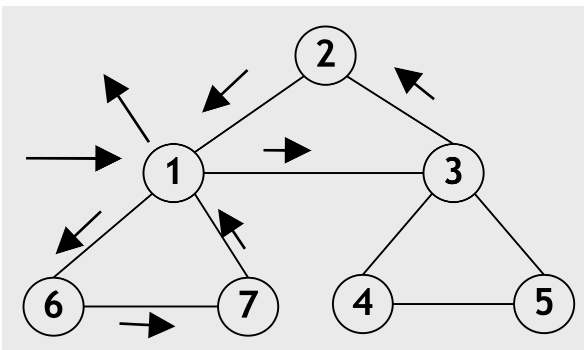
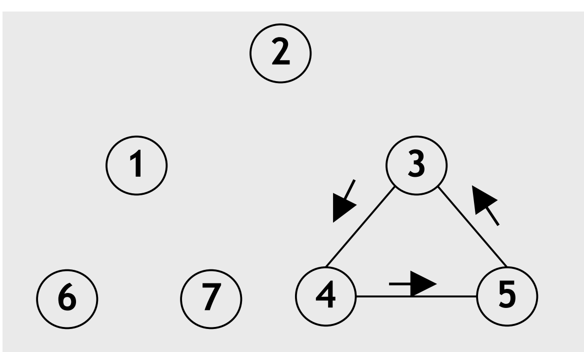
2. Enquanto existirem arestas por visitar

   2.1 Procurar o primeiro vértice no caminho (circuito) obtido até ao momento que possua uma aresta não percorrida

   2.2 Lançar uma sub-pesquisa em profundidade a partir desse vértice (sem voltar a percorrer arestas já percorridas)

   2.3 Inserir o resultado (circuito) no caminho principal

Exemplo em grafo não dirigido

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

Estruturas de dados e eficiência temporal

- Tempo de execução: $O(|E| + |V|)$
  - Cada vértice e aresta é percorrido uma única vez
  - Cada vez que se percorre um adjacente, avança-se o apontador de adjacentes (para não voltar a percorrer as mesmas arestas)
  - Usam-se listas ligadas para efetuar inserções em tempo constante

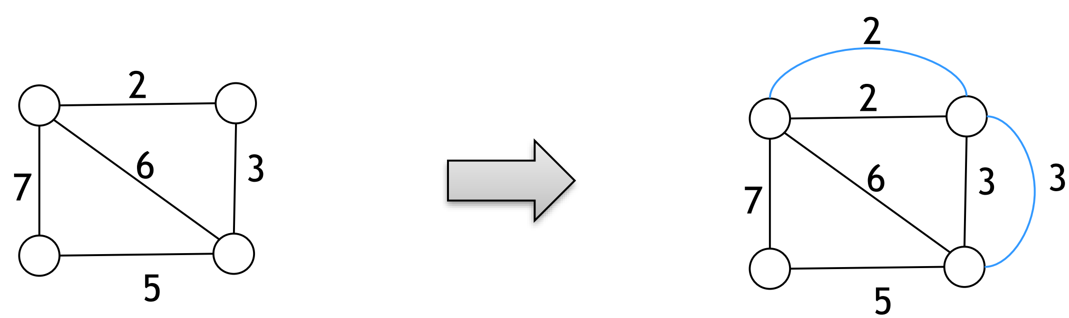
Problema do carteiro chinês (*Chinese postman problem*)

- Dado um grafo pesado conexo G=(V,E), encontrar um caminho fechado (i.e., com início e fim no mesmo vértice) de peso mínimo que atravesse cada aresta de *G* pelo menos uma vez.
  - A um caminho 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*.
- Problema estudado pela primeira vez por Mei-Ku Kuan em 1962, relacionado com a distribuição de correspondência ao longo de um conjunto de ruas, partindo e terminando numa estação de correios.
- Resolúvel em tempo polinomial para grafos dirigidos ou não dirigidos, mas infelizmente o problema é NP-completo (tempo exponencial) quando se combinam arestas dirigidas com arestas não dirigidas (grafos mistos)
  - Exemplo: percurso do camião do lixo, quando algumas ruas têm sentidos únicos

Abordagem

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

Não Euleriano (peso 23)

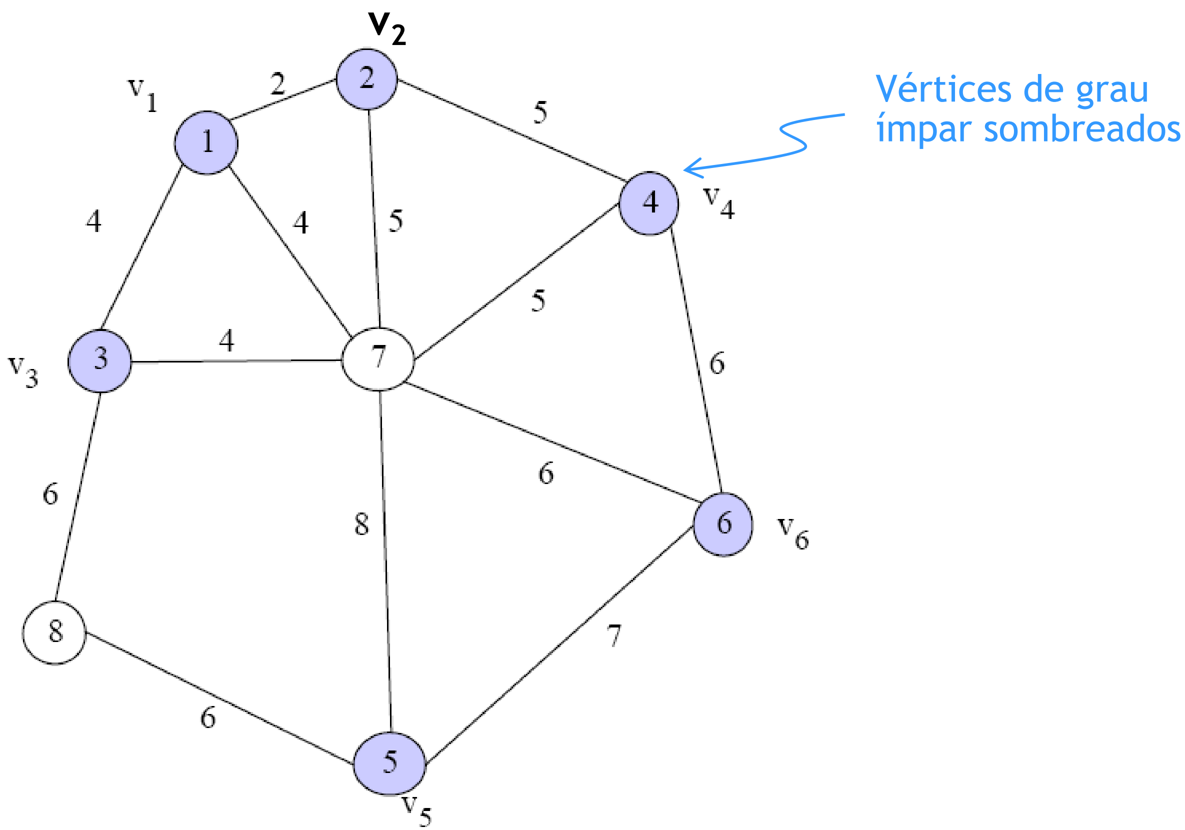


Euleriano! (peso 28)

Método para grafos não dirigidos (1/4)

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

G:



Método para grafos não dirigidos (2/4)

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

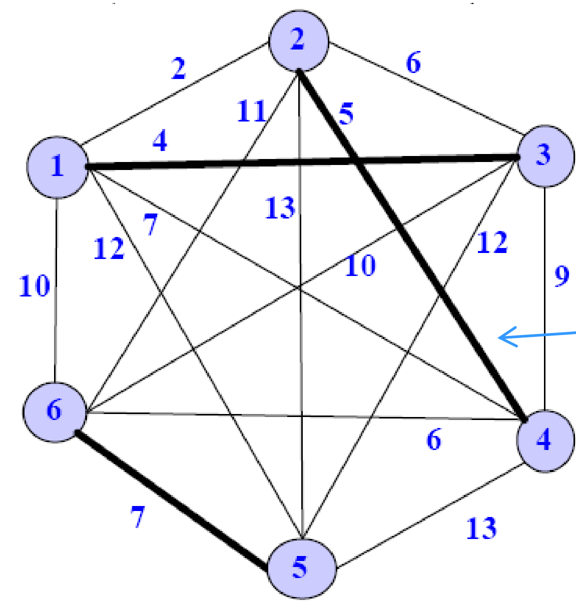
$d(v_i,v_j)$	v1	v2	v3	v4	v5	v6
v1	-	2	4	7	12	10
v2		-	6	5	13	11
v3			-	9	12	10
v4				-	13	6
v5					-	7
v6						-

Método para grafos não dirigidos (3/4)

- Passo 3: Construir um grafo completo *G'* com os vértices de grau ímpar de *G* ligados entre si por arestas de peso igual à distância mínima calculada no passo 2.

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

G':

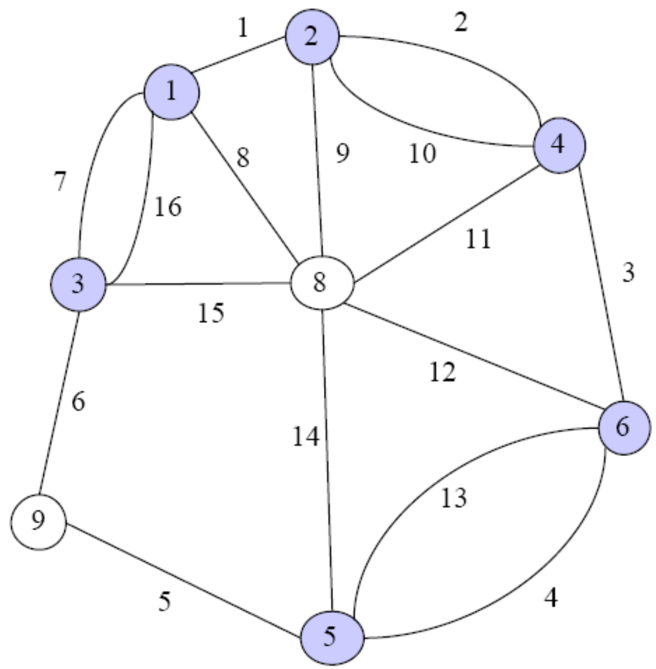


emparelhamento
indicado a traço
forte

Método para grafos não dirigidos (4/4)

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

G*:



Nº de sequência de visita no circuito Euleriano (pesos omitidos)

* Realização do passo 4 - Emparelhamento perfeito de peso mínimo

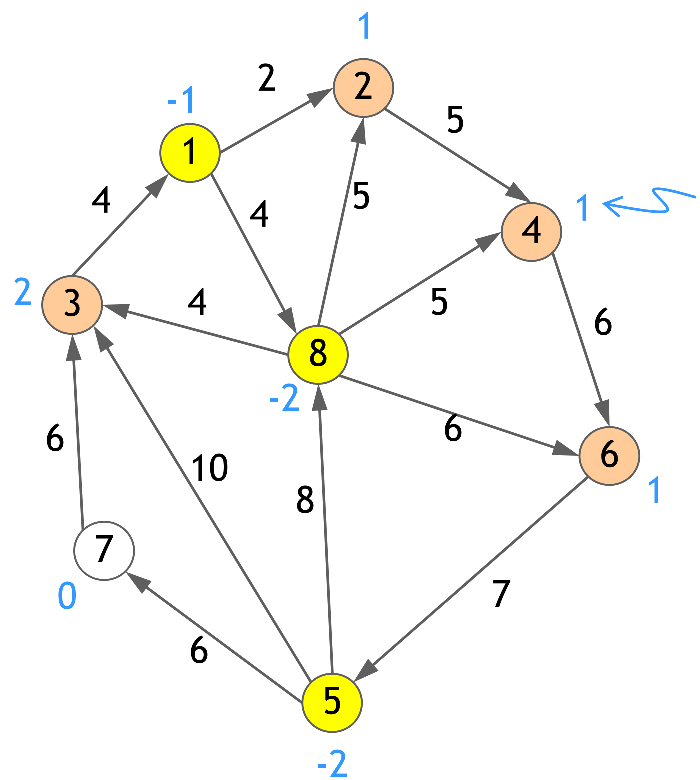
- O problema de encontrar um emparelhamento perfeito de peso mínimo pode ser reduzido ao problema de encontrar um emparelhamento de peso máximo num grafo genérico por uma simples mudança de pesos
  - Basta substituir cada peso w_{ij} por $M+1-w_{ij}$, *em* que M é o peso da aresta mais pesada
  - Sendo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

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

Método para grafos dirigidos (1/4)

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

G:

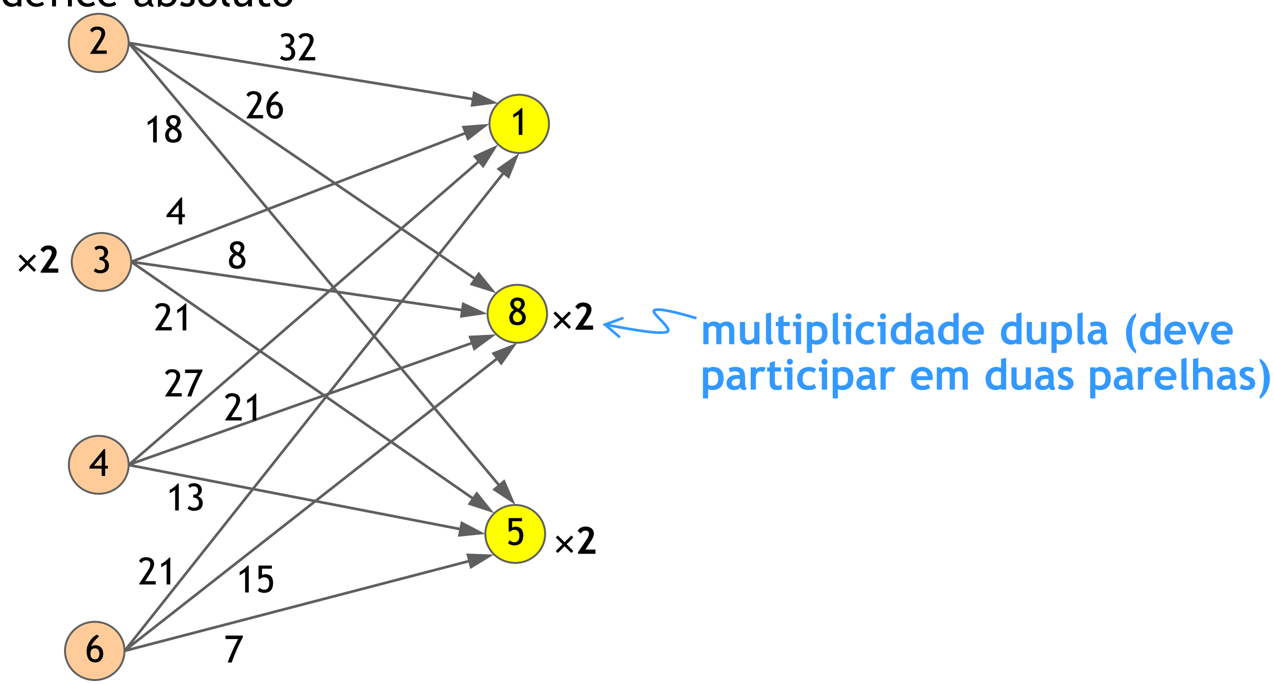


(nº de entradas) - (nº de saídas)

Método para grafos dirigidos (2/4)

- Passo 2: Determinar os caminhos mais curtos de vértices que têm défice de saídas para vértices que têm défice de entradas e representar as distâncias respetivas num grafo bipartido *G'*.
  - Vértices são anotados com multiplicidade (nº de parelhas em que deve participar) igual ao défice absoluto

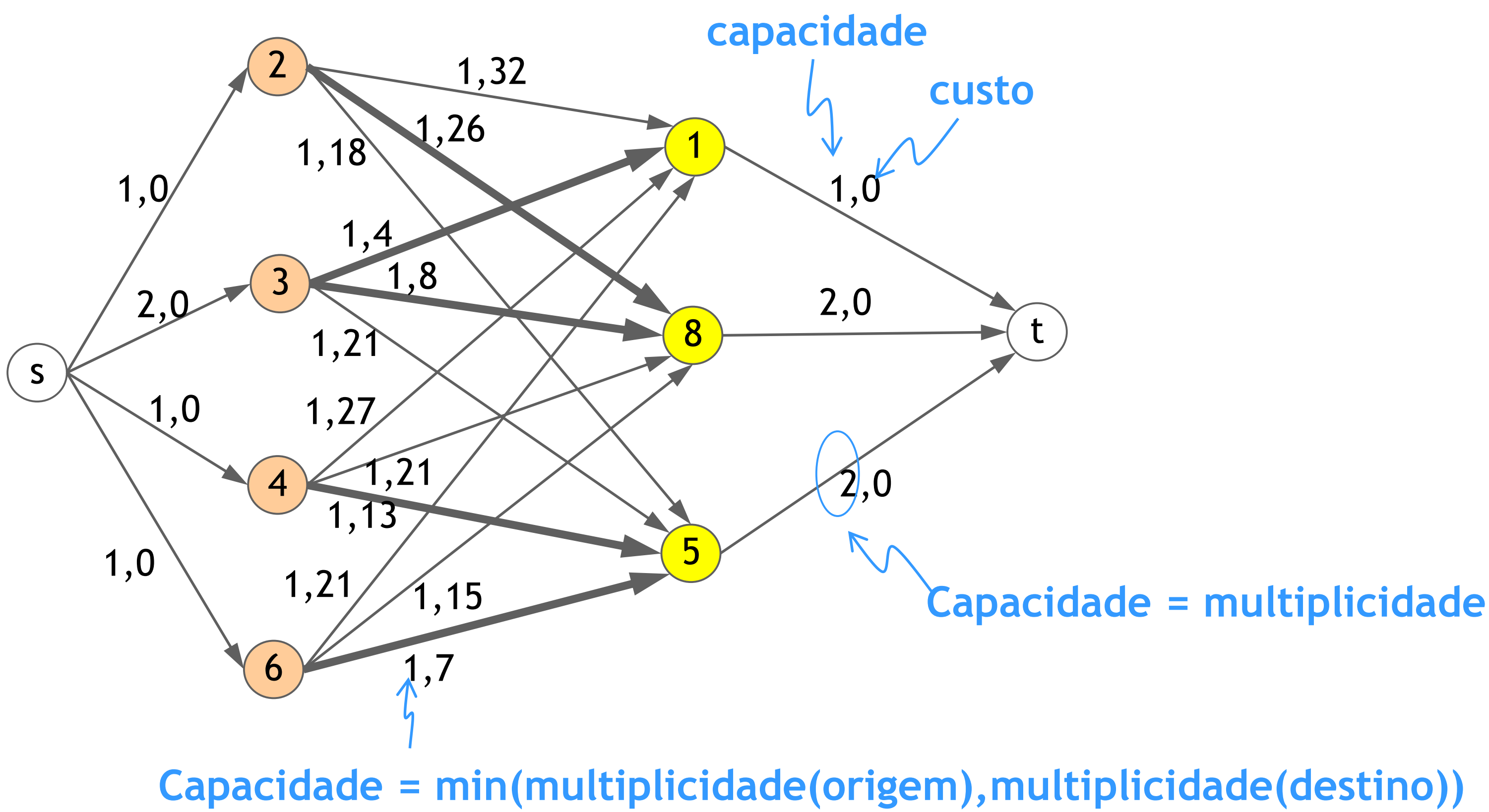
G':



Método para grafos dirigidos (3/4)

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

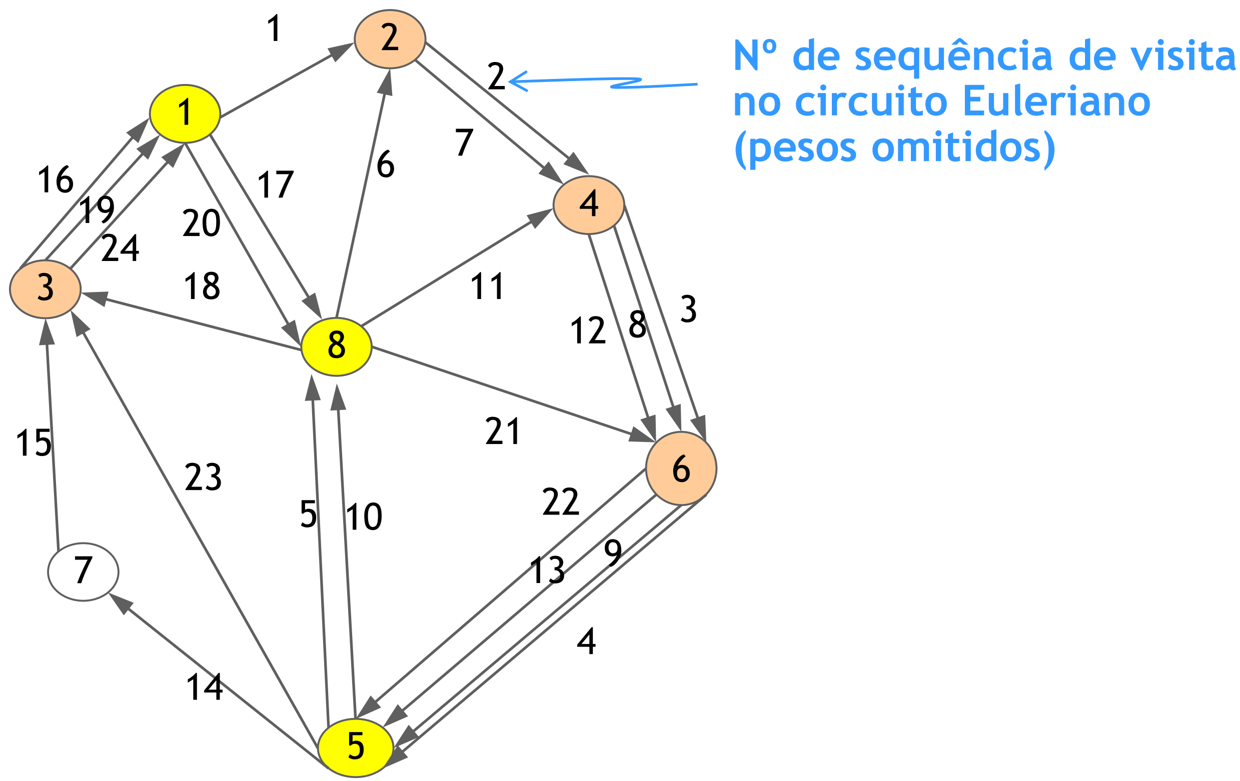
G'':



Método para grafos dirigidos (4/4)

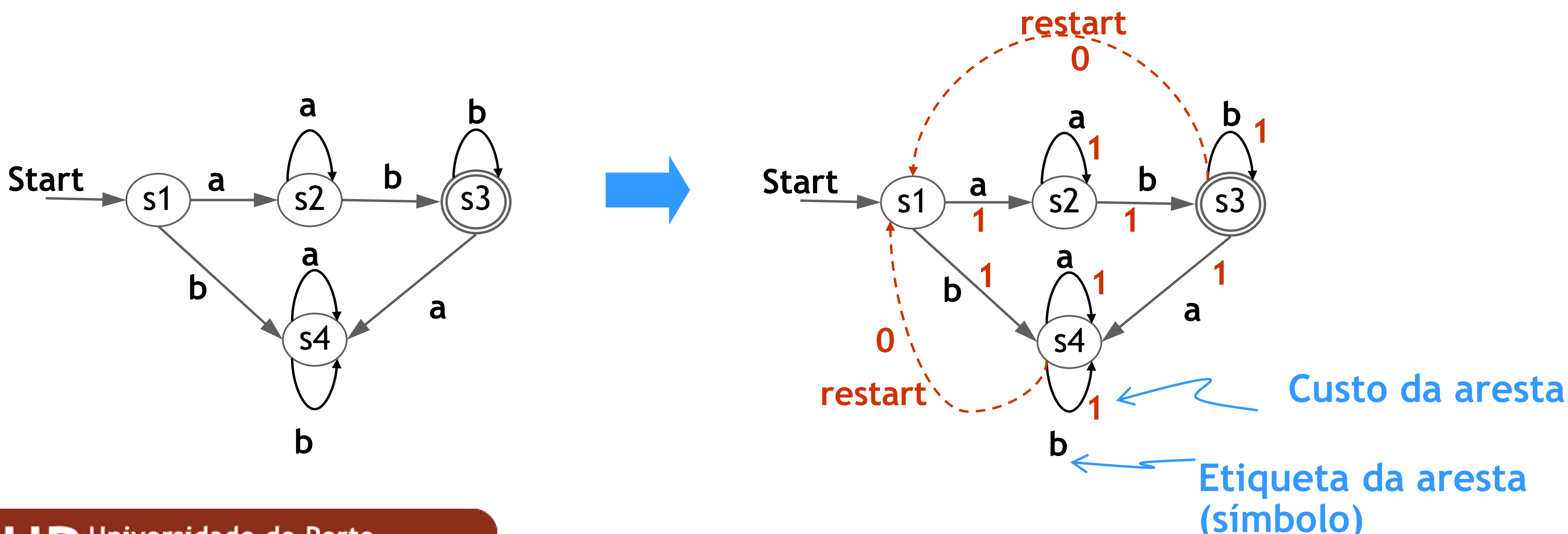
- Passo 4: Obter grafo Euleriano *G**, duplicando em *G* os caminhos mais curtos entre os vértices emparelhados no passo 3, e obter um circuito Euleriano.

G*:



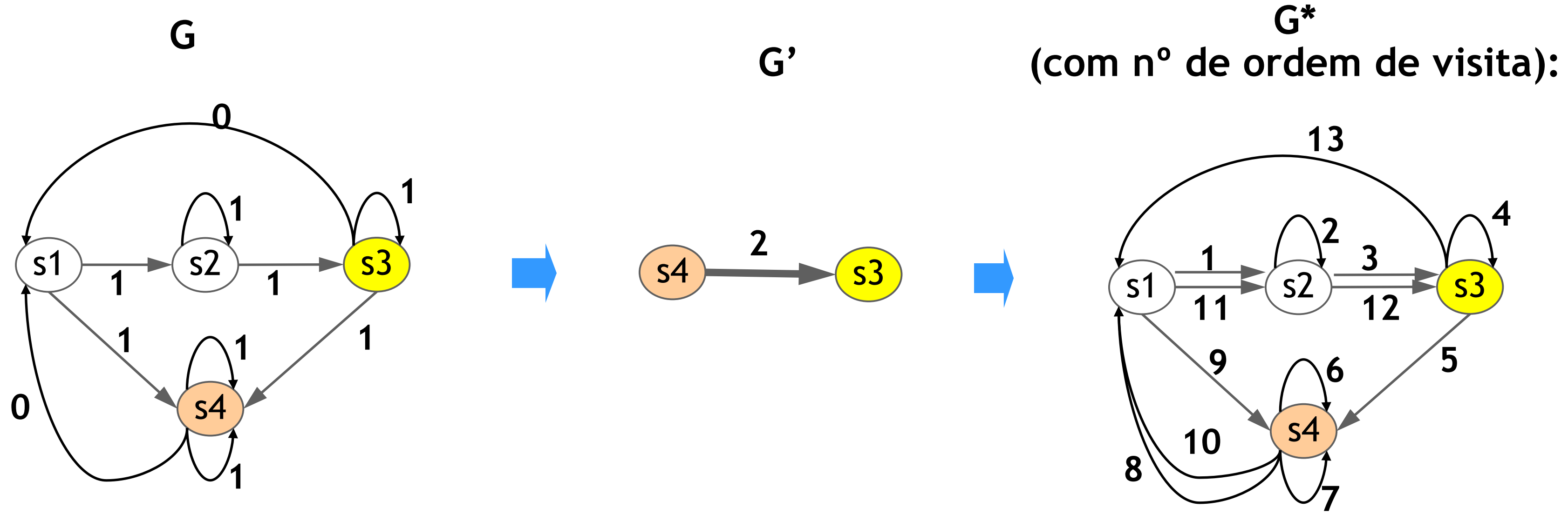
* Exemplo de aplicação (1/2)

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



* Exemplo de aplicação (2/2)

- Resolução do problema do carteiro chinês dirigido:



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

Referências e mais informação

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

Algoritmos de Pesquisa em Strings

Rosaldo Rossetti

Desenho de Algoritmos, L.EIC

Índice

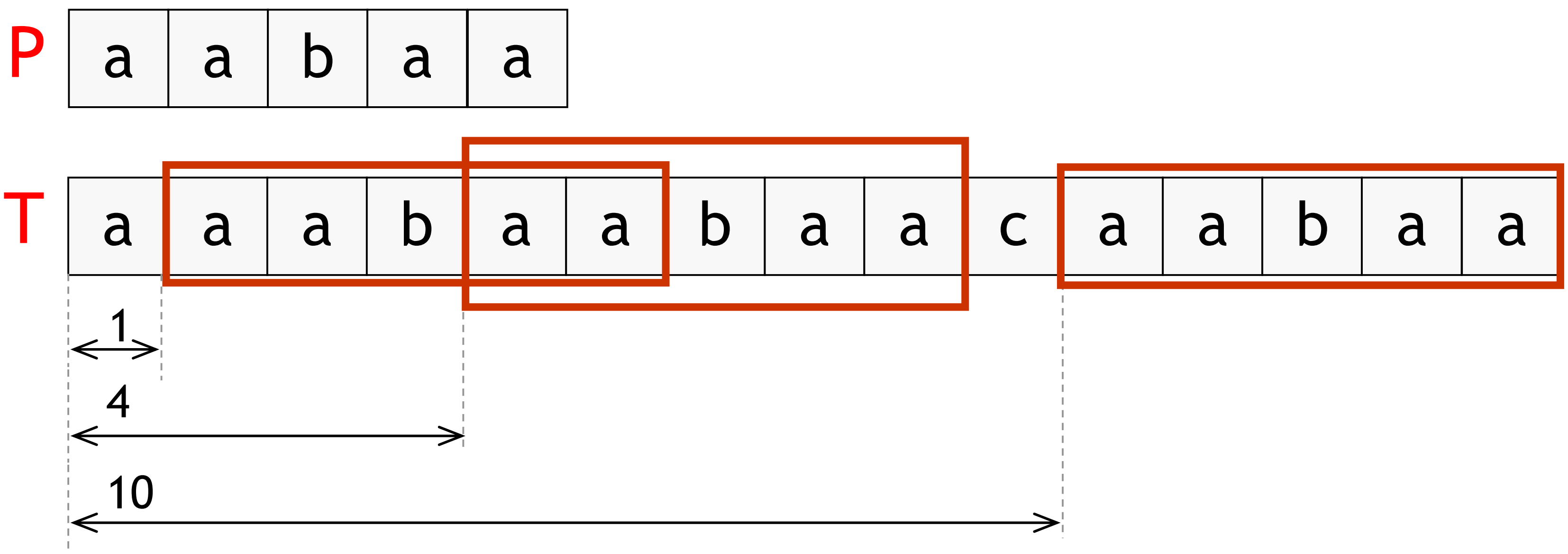
- Pesquisa exata (*string matching*)
- Pesquisa aproximada (*approximate string matching*)
  - Distância de edição (Programação Dinâmica)

Pesquisa exata

(*string matching*)

Problema

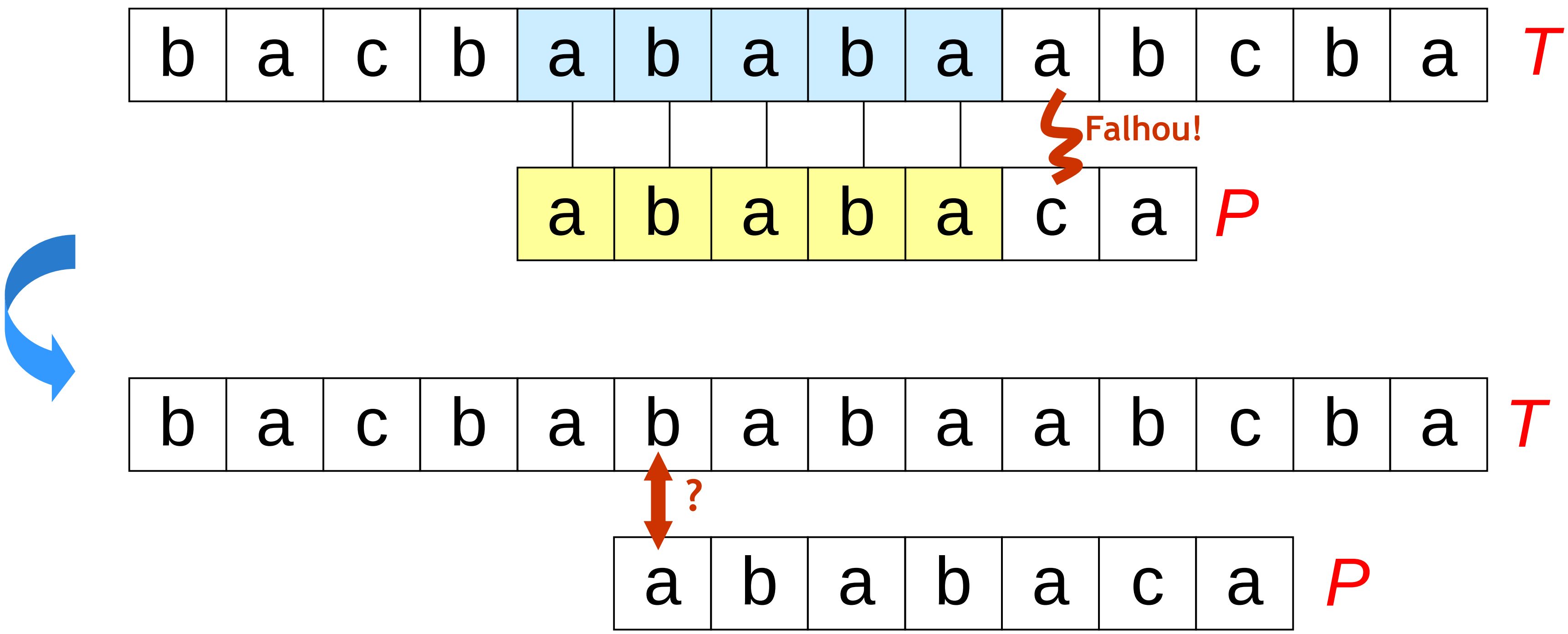
- Encontrar todas as ocorrências de um padrão P num texto T
    - P e T são cadeias de caracteres
    - Ocorrências são definidas pela deslocação em relação ao início do texto
    - Ocorrências podem ser sobrepostas



Algoritmos

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

Algoritmo *naive*

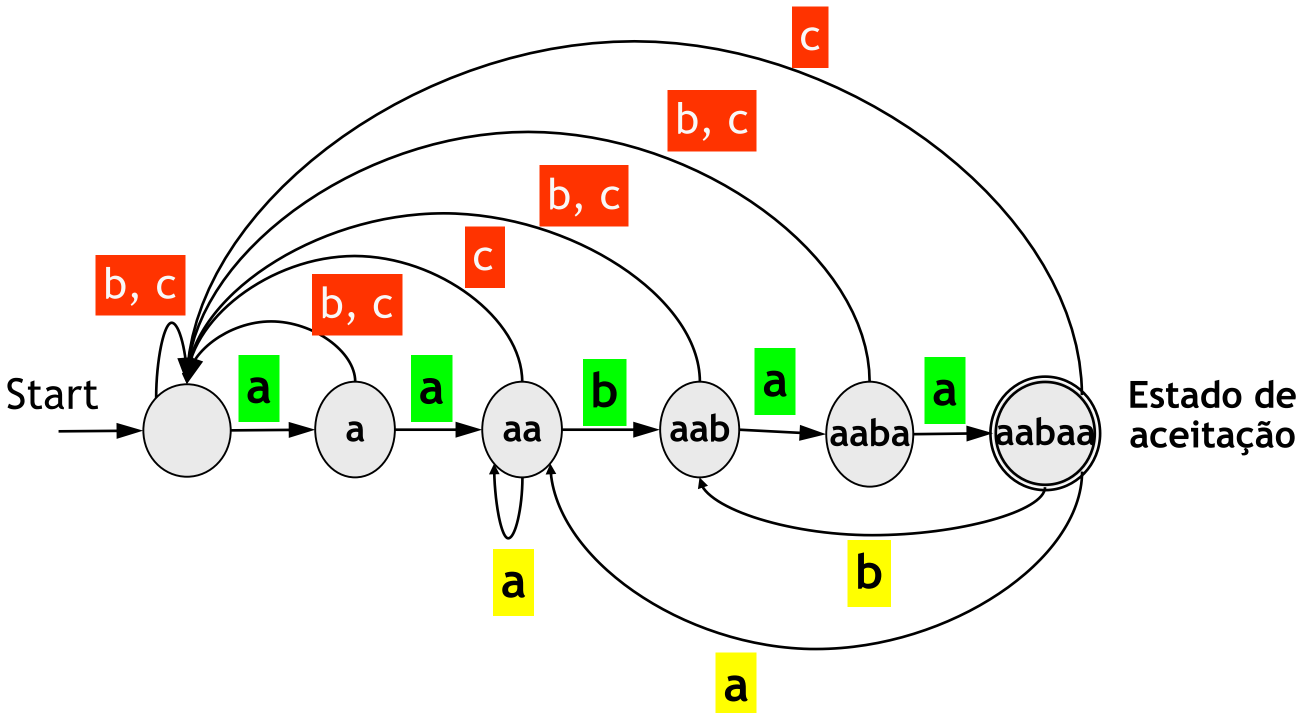


Desloca-se o padrão uma casa para a direita e recomeça-se a comparação do início do padrão!   Ineficiente: O(|P| |T|)

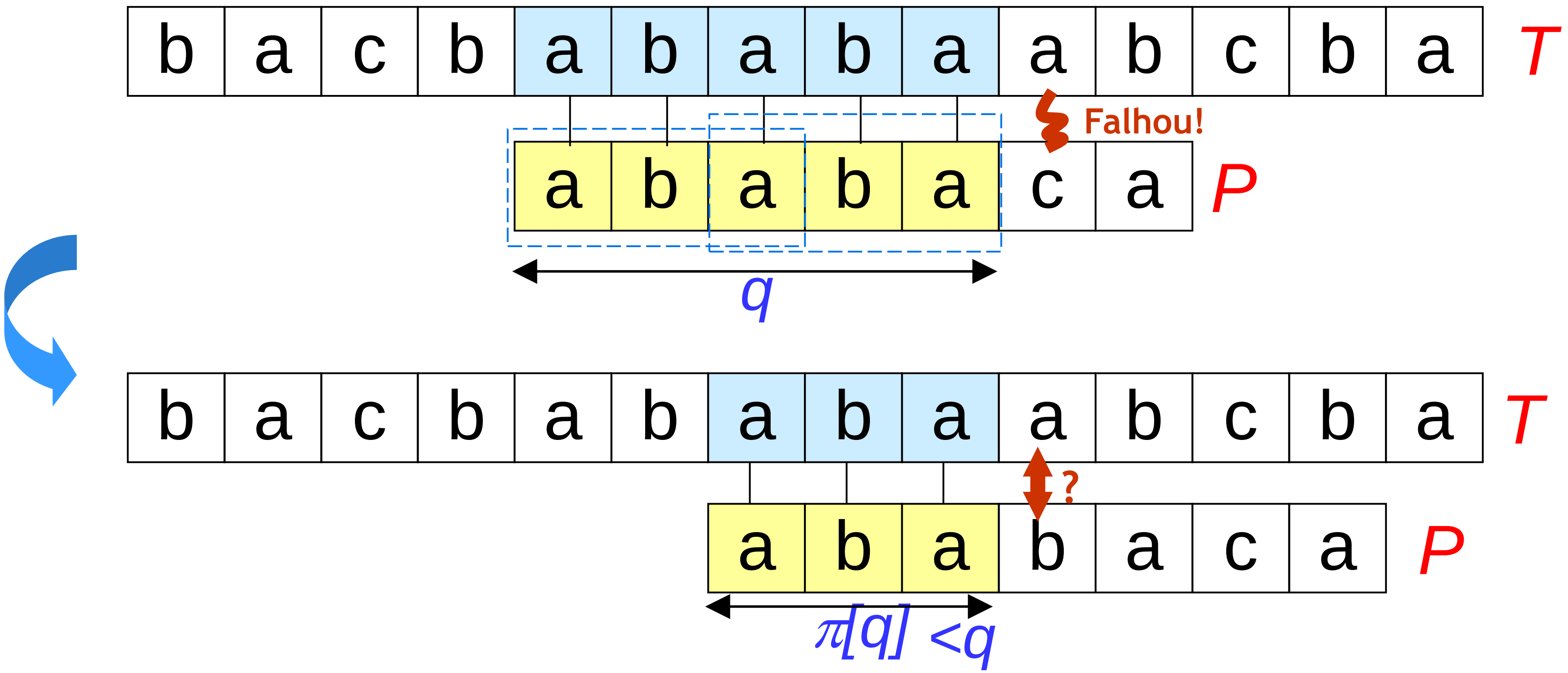
Autómato finito correspondente ao padrão

P	a	a	b	a	a

$\Sigma = \{a, b, c\}$



Algoritmo de Knuth-Morris-Pratt



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

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

Pré-processamento do padrão

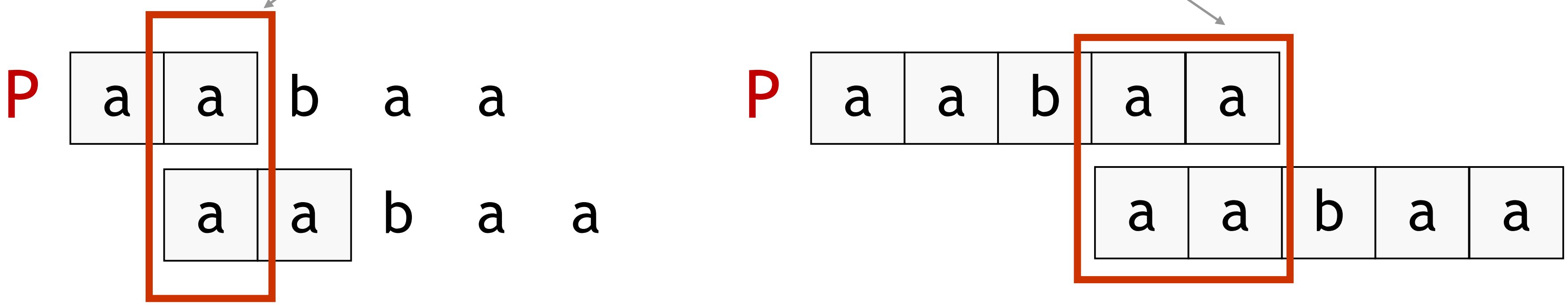
- Compara-se o padrão com deslocações do mesmo, para determinar a **função prefixo**

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

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

Pré-processamento do padrão

q	1	2	3	4	5
P[q]	a	a	b	a	a
π[q]	0	1	0	1	2



Pseudo-código

```
KMP-MATCHER(T, P)
n ← length[T]
m ← length[P]
π ← COMPUTE-PREFIX-FUNCTION(P)
q ← 0                                        ▷ Number of characters matched.
for i ← 1 to n                               ▷ Scan the text from left to right.
     do while q > 0 and P[q + 1] ≠ T[i]
             do q ← π[q]                     ▷ Next character does not match.
        if P[q + 1] = T[i]
          then q ← q + 1                     ▷ Next character matches.
        if q = m                             ▷ Is all of P matched?
          then print "Pattern occurs with shift" i − m
                q ← π[q]                     ▷ Look for the next match.
```

Pseudo-código

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
     do while k > 0 and P[k + 1] ≠ P[q]
            do k ← π[k]
        if P[k + 1] = P[q]
          then k ← k + 1
        π[q] ← k
return π
```

* Eficiência do algoritmo Knuth-Morris-Pratt

- KMP-MATCHER (sem incluir COMPUTE-PREFIX-FUNCTION)
  - Eficiência depende do nº de iterações do ciclo "while" interno
  - Dado que $0 \leq \pi[q] < q$, cada vez que a instrução 7 é executada, o valor de q é decrementado de pelo menos 1, sem nunca chegar a ser negativo
  - Dado que o valor de q começa em 0 e só é incrementado no máximo n vezes (+1 de cada vez, na linha 9), o nº máximo de vezes que pode ser decrementado (nas linhas 7 e 12) é também n

  $\Rightarrow$ Nº máximo de iterações do ciclo "while" interno (no conjunto de todas as iterações do ciclo "for" externo) é n

  $\Rightarrow$ Tempo de execução da rotina é $O(n)$, i.e., $O(|T|)$
- COMPUTE-PREFIX-FUNCTION
  - Seguindo o mesmo raciocínio, tempo de execução é $O(m)$, i.e., $O(|P|)$
- Total: O(n+m),isto é, **O(|T| + |P|)**

Referências e mais informação

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