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

R. Rossetti, A. P. Rocha, J. Pascoal Faria

FEUP, MIEIC, CAL, 2013/2014

Nota prévia

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

Índice

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

* Emparelhamentos

Conceito de emparelhamento

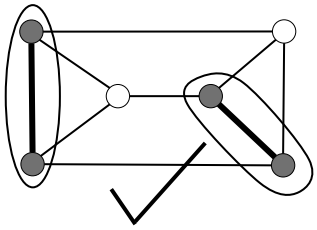
- Seja o grafo **não dirigido** $G = (V, E)$
- Formalmente, um emparelhamento (*matching*) M em G é um conjunto de arestas que não contém mais do que uma aresta incidente no mesmo vértice
- Ou: um conjunto de arestas independentes num grafo G é um conjunto de arestas sem vértices comuns!

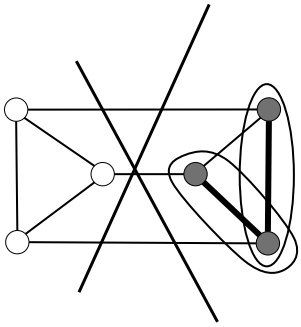
Alguns conceitos

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

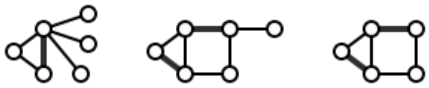
Conceito de emparelhamento

- Exemplo:

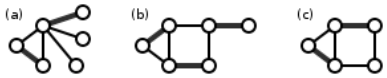




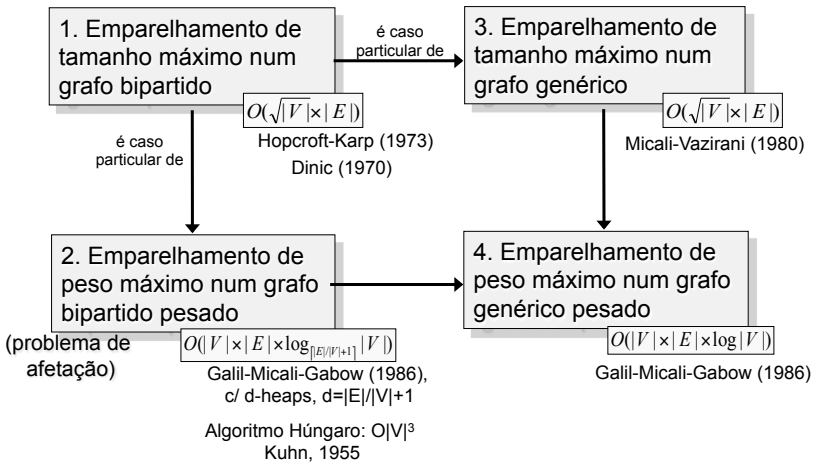
•Emparelhamento Maximal



•Emparelhamento Máximo

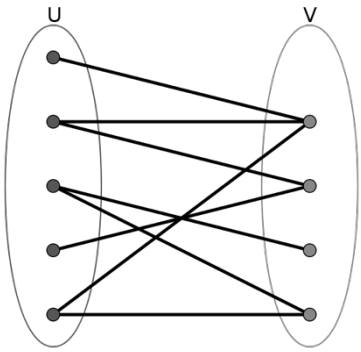


Problemas de emparelhamento



Grafos bipartidos

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

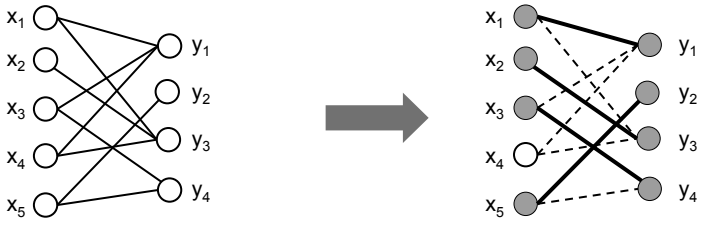


Redução a problemas em redes de transporte

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

1. Emparelhamento de tamanho máximo num grafo bipartido

e.g. pessoas candidatam-se a empregos

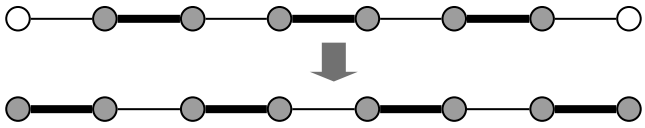


Grafo *bipartido* - o conjunto de vértices pode ser partido em dois conjuntos *X* e *Y* tal que todas as arestas têm um extremo em *X* e outro em *Y*

Quais são as outras soluções?

* Método dos caminhos de aumento (*augmenting paths*)

- Definição: Um caminho simples *P* (aberto ou fechado) num grafo *G* (bipartido ou genérico) é um *caminho de aumento* em relação a um emparelhamento *M* se
  (a) começa e acaba em vértices que não estão em *M* (vértices livres), e
  (b) passa alternadamente em arestas que não pertencem e pertencem a *M*
- Removendo de *M* as arestas de *P* que pertencem a *M*, e adicionando as que não pertencem a *M*, aumenta-se o tamanho do emparelhamento de 1

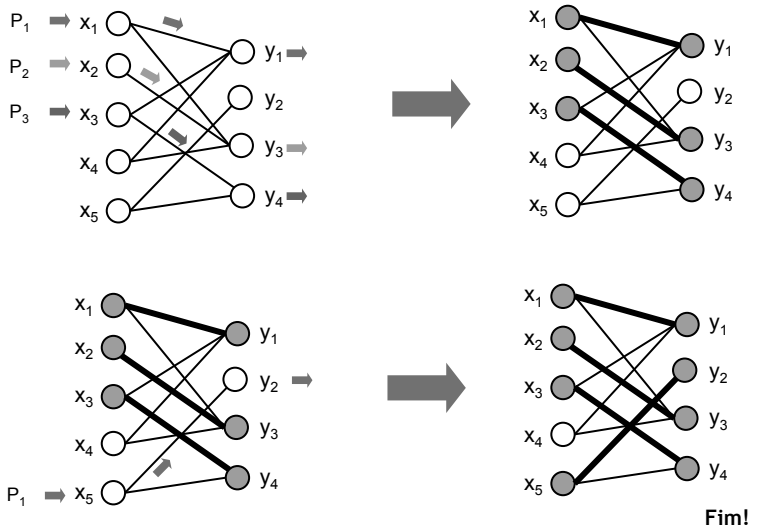


- A aplicação sucessiva desta técnica, partindo de um emparelhamento vazio, até não existirem mais caminhos de aumento, dá um emparelhamento de tamanho máximo (num grafo bipartido ou genérico), de acordo com o …
- Teorema de Berge: Um emparelhamento *M* num grafo *G* (bipartido ou genérico) tem tamanho máximo sse não existir nenhum caminho de aumento

* Algoritmo de Hopcroft-Karp (1973)

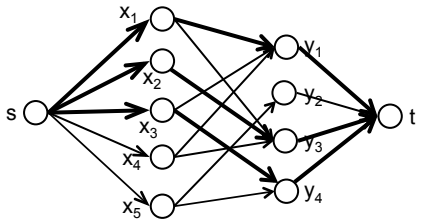
- Algoritmo:
  1. M ← { }
  2. Encontrar um conjunto maximal de caminhos de aumento de comprimento mínimo, sem vértices em comum
     - Um conjunto maximal é um conjunto a que não é possível acrescentar mais elementos
     - Minimizando o comprimento dos caminhos, maximiza o nº de caminhos e portanto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

* Exemplo de aplicação do algoritmo de Hopcroft-Karp

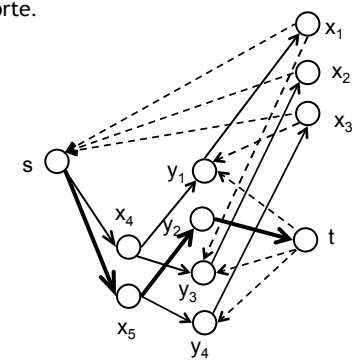


Resolução como problema de fluxo máximo numa rede unitária (Dinic)

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



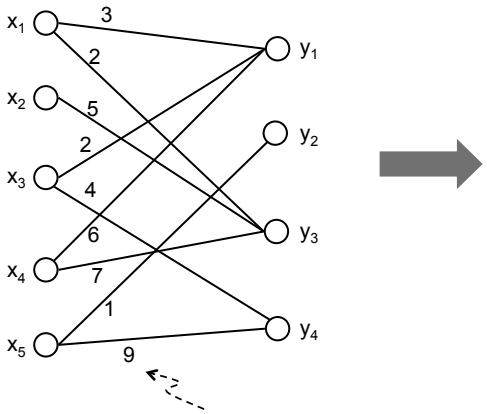
Novo grafo de resíduos, ordenado por níveis da esquerda para a direita.
Caminho de aumento marcado a traço forte.



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

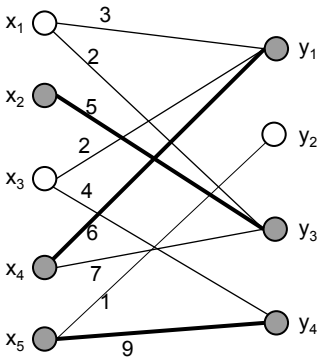
2. Emparelhamento de peso máximo num grafo bipartido pesado (problema de afetação)

e. g. pessoas candidatam-se a empregos



preferência, aptidão, etc.

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



Peso total: 20

(estratégia gananciosa daria 19)

* Representação tabular (matriz de adjacências)

	y1	y2	y3	Y4
x1	3	-	2	-
x2	-	-	5	-
x3	2	-	-	4
x4	6	-	7	-
x5	-	1	-	9

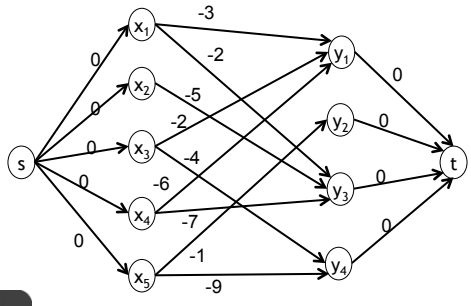
Pode-se colocar um valor negativo (-1) para garantir que não é selecionado

Mais adequada quando o grafo é denso!

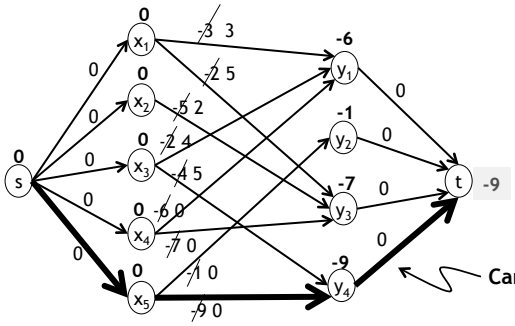
Se o grafo for denso, um bom algoritmo é o algoritmo Húngaro!

* Resolução como problema de fluxo de custo mínimo

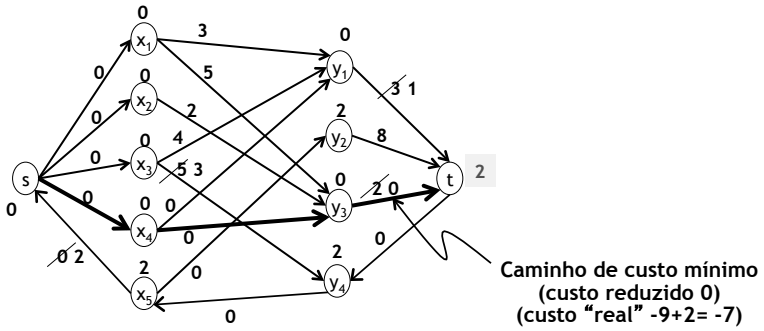
- Capacidades unitárias (como tal apenas se mostram os custos e não as capacidades)
- Faz-se custo no problema de transporte = simétrico do peso no problema de emparelhamento
  - Origina arestas de custo negativo, mas não há ciclos
- Aplica-se método dos caminhos de aumento de custo mínimo, parando-se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



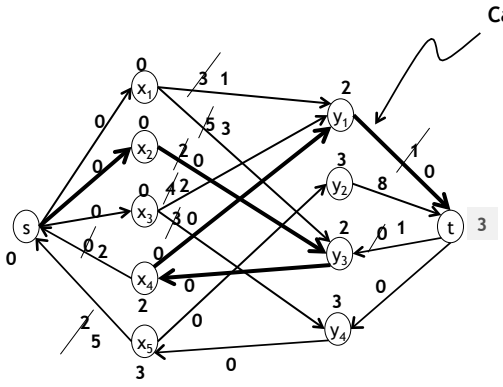
* Resolução como problema de fluxo de custo mínimo – 1ª iteração



* Resolução como problema de fluxo de custo mínimo – 2ª iteração

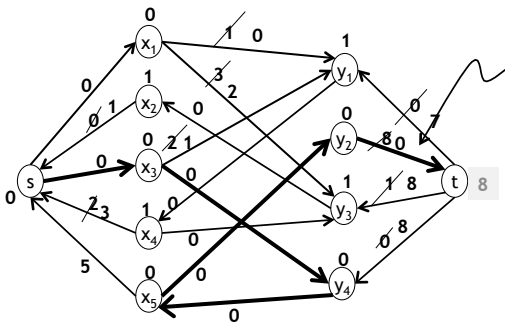


* Resolução como problema de fluxo de custo mínimo – 3ª iteração



Caminho de custo mínimo
(custo reduzido 0)
(custo "real" -7+3= -4)

* Resolução como problema de fluxo de custo mínimo – 4ª iteração



Caminho de custo mínimo
(custo reduzido 0)
(custo "real" -4+8= 4 >= 0)

PARAR (sem aplicar este aumento)

Dá a solução esperada!

* Eficiência da resolução como problema de fluxo de custo mínimo

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

3. Emparelhamento de tamanho máximo num grafo genérico

Quantas soluções há?

* Generalização do método dos caminhos de aumento para grafos genéricos

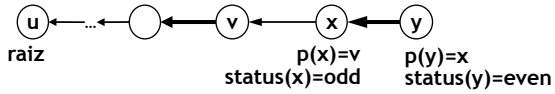
- (ver bibliografia)

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

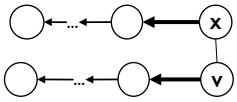
- Ideia geral: Construir uma floresta de visita do grafo ao longo de *"caminhos alternados"* (formados alternadamente por arestas livres/ emparelhadas) partindo dos vértices livres (cada vértice livre é raiz duma árvore), até encontrar uma aresta que liga duas árvores formando um caminho de aumento válido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

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

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

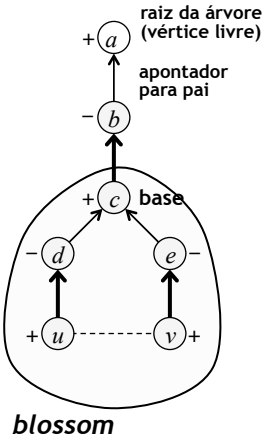


  - Se *status(x)=even* (como o grafo é bipartido, implica que v e x fazem parte de árvores diferentes), está encontrado um caminho de aumento entre as raízes das duas árvores passando por esta aresta (v,x)
  - Nos outros casos, ignorar a aresta



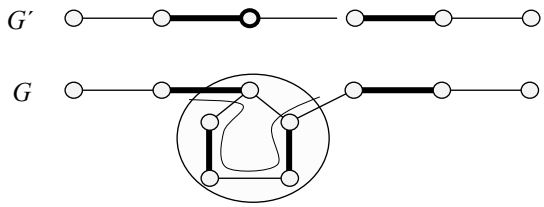
* *Blossoms*

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



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

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



* Algoritmo de Edmonds-Karp (1/5)

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

* Algoritmo de Edmonds-Karp (2/5)

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

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

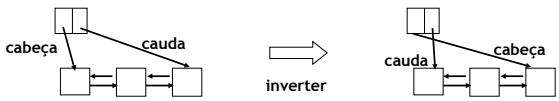
- Se *status(w)* = *unreached*, prolongar o caminho
  - Prolongar o caminho de visita corrente por *w* e *mate(w)*, fazendo
    root(w) ← *root(v)*, *status(w)* ← *odd*, *p(w)* ← *v*,
    root(mate(w)) ← *root(v)*, *status(mate(w))* ← *even*, *p(mate(w))* ← *w*
  - Acrescentar *mate(w)* à fila *q*
-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

* Algoritmo de Edmonds-Karp (4/5)
Definição recursiva de nca(x,y)

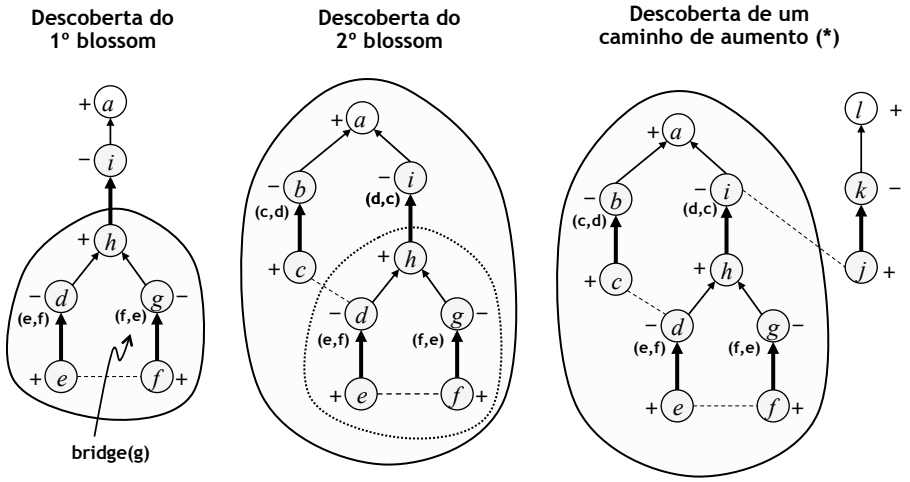
- Caminha-se no sentido da raiz da árvore em paralelo a partir dos dois vértices (*v* e *w*), até encontrar por um caminho um vértice já visitado pelo outro caminho
- Usa-se uma flag auxiliar *visited* para marcar os vértices já visitados, a qual é limpa no caminho de regresso para os vértices de partida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

* Algoritmo de Edmonds-Karp (5/5)
Definição recursiva de path(x,y)

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



* Exemplo



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

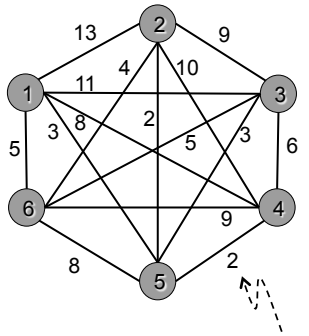
* Eficiência do algoritmo de Edmonds-Karp

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

*Algoritmo de Micali-Vazirani (1980)

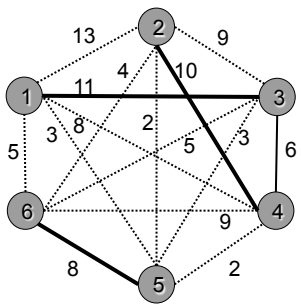
- Modifica o algoritmo de Edmonds-Karp, por forma a conseguir encontrar numa única passagem um conjunto maximal de caminhos de aumento disjuntos, conforme acontece no algoritmo de Hopcroft-Karp para o caso bipartido
- Consegue atingir a mesma eficiência que o algoritmo de Hopcroft-Karp, ou seja, $O(\sqrt{|V|}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

4. Emparelhamento de peso máximo num grafo genérico



preferência dos alunos 4 e 5 para trabalharem juntos

Neste exemplo o grafo é completo



Peso total: 29

(estratégia gananciosa dava 25)

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

* Algoritmos para determinação de emparelhamentos de peso máximo

- (ver bibliografia)

* Problema dos casamentos estáveis (*stable marriage*)

Problema

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

Algoritmo de Gale-Shapley (1962)

Algoritmo de Gale-Shapley (1962)

Considerar inicialmente que todas as pessoas estão livres.
Enquanto houver algum homem h livre fazer:
    seja m a primeira mulher na lista de h a quem este ainda não se propôs;
    se m estiver livre então
        emparelhar h e m (ficam noivos)
    senão
        se m preferir h ao seu actual noivo h' então
            emparelhar h e m (ficam noivos), voltando h' a estar livre
        senão
            m rejeita h e assim h continua livre.
fim

Tempo de execução: $O(n^2)$

Listas de preferências incompletas

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

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

  Caso contrário, diz-se estável.

* Algoritmo de Gale-Shapley com listas de preferências incompletas

ALGORITMO DE GALE-SHAPLEY (ORIENTADO POR INTERNOS)

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

* Propriedades do algoritmo de Gale-Shapley

- O emparelhamento obtido por este algoritmo é ótimo para os internos e péssimo para os hospitais: qualquer interno fica com o melhor hospital que pode ter em qualquer emparelhamento estável e cada hospital fica com os piores internos.
- Tempo de execução é de ordem quadrática (nº de internos * nº de hospitais)

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

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

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

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

Referências e mais informação

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