

Teoria dos Grafos - Notas de Aula

Estrutura do curso

Parte 1: Conceitos e fundamentos

Parte 2: Algoritmos em grafos

a) Problemas simples (P): soluções ótimas, algoritmos eficientes

b) Problemas complexos (NP): soluções aproximadas, algoritmos ineficientes

Bibliografia

A First Look at Graph Theory. John Clark & Derek Allan Holton, World Scientific, 1998.

Fundamentos da Teoria dos Grafos para Computação. Nicoletti, M.C.; Hruschka Jr., E. R., 2 ed., Série Apontamentos, EdUFSCar, 2009.

Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest and Clifford Stein, 3 ed., The MIT Press, 2009.

Ambiente virtual: <https://ava.ead.ufscar.br/course/view.php?id=3179>

Cômputo da média final

MF = 0.25 x P1 + 0.25 x P2 + 0.25 x P3 + 0.05 x T1 + 0.05 x T2 + 0.05 x T3 + 0.05 x T4 + 0.05 x T5

P1, P2 e P3: avaliações presenciais 1, 2 e 3

T1, T2, T3, T4 e T5: notas dos trabalhos práticos (cada projeto vale 0.5 pontos na média final)

Critério para aprovação: MF \geq 6.0 e frequência \geq 75%

Horário de atendimento:

Terça das 13hs as 15hs

Estruturas Discretas: uma breve revisão

Relação Binária em A

Seja A um conjunto qualquer. R é uma relação binária em A se R é um subconjunto do produto cartesiano A x A, ou seja $R \subseteq A \times A$

Ex: $A = \{1, 2, 3\}$

$$\begin{aligned} A \times A &= \{\langle 1, 1 \rangle, \langle 1, 2 \rangle, \langle 1, 3 \rangle, \langle 2, 1 \rangle, \langle 2, 2 \rangle, \langle 2, 3 \rangle, \langle 3, 1 \rangle, \langle 3, 2 \rangle, \langle 3, 3 \rangle\} \\ R &= \{\langle 1, 2 \rangle, \langle 3, 3 \rangle\} \end{aligned}$$

Obs: Para dizer que um par ordenado $\langle x, y \rangle$ pertence a uma relação R, temos que xRy

Relações possuem propriedades intrínsecas. Iremos listar algumas das mais importantes:

- 1) Reflexiva: $\forall x \in A (xRx)$
- 2) Simétrica: $\forall x, y \in A (xRy \rightarrow yRx)$
- 3) Anti-simétrica: $\forall x, y \in A ((xRy \wedge yRx) \rightarrow x = y)$
- 4) Transitiva: $\forall x, y, z \in A ((xRy \wedge yRz) \rightarrow xRz)$

Ex: $A = \{1, 2, 3\}$

	R	S	AS	T
$R_1 = \{\langle 1, 2 \rangle, \langle 2, 2 \rangle, \langle 3, 2 \rangle, \langle 2, 3 \rangle\}$	F	F	F	F
$R_2 = \{\langle 2, 3 \rangle\}$	F	F	V	V
$R_3 = \{\langle 3, 3 \rangle\}$	F	V	V	V

Relações de Equivalência

R é uma relação de equivalência se R satisfaz as propriedades:

- Reflexiva
- Simétrica
- Transitiva

Ex: Congruência em módulo m

$$a \equiv_m b \Leftrightarrow (a - b) \bmod m = 0$$

Diremos que a é m-congruente a b se o resto da divisão de $(a - b)$ por m resultar em zero.

Considere o conjunto a seguir

$$A = \{-2, -1, 0, 1, 2, 3, 4\}$$

$$\begin{aligned} \equiv_3 &= \{\langle -2, -2 \rangle, \langle -1, -1 \rangle, \langle 0, 0 \rangle, \langle 1, 1 \rangle, \langle 2, 2 \rangle, \langle 3, 3 \rangle, \langle 4, 4 \rangle \\ &\quad \langle -2, 1 \rangle, \langle -2, 4 \rangle, \langle -1, 2 \rangle, \langle 0, 3 \rangle, \langle 1, -2 \rangle, \langle 1, 4 \rangle, \langle 2, -1 \rangle, \langle 3, 0 \rangle, \langle 4, -2 \rangle, \langle 4, 1 \rangle\} \end{aligned}$$

Definição: A classe de equivalência de um elemento $x \in A$ é o conjunto de todos os elementos que estão relacionados a ele, sendo denotada por

$$R[x] = \{y / xRy\}$$

No caso do exemplo acima temos as seguintes classes de equivalência

$$\begin{aligned} R[-2] &= \{-2, 1, 4\} \\ R[-1] &= \{-1, 2\} \end{aligned}$$

$R[0] = \{0, 3\}$
 $R[1] = \{1, -2, 4\}$
 $R[2] = \{2, -1\}$
 $R[3] = \{3, 0\}$
 $R[4] = \{4, -2, 1\}$

Prop: As classes de equivalência induzem uma partição no conjunto A.

Partição de um conjunto P: Subdivisão de P em subconjuntos P_1, P_2, \dots, P_n tais que
 $P_1 \cup P_2 \cup \dots \cup P_n = P$ e $P_1 \cap P_2 \cap \dots \cap P_n = \emptyset$

Ex: $A_1 = \{-2, 1, 4\}$
 $A_2 = \{-1, 2\}$
 $A_3 = \{0, 3\}$

Questão: Suponha que R é uma relação sobre o conjunto das palavras, de modo que xRy se e somente se $\text{tam}(x) = \text{tam}(y)$, onde $\text{tam}(x)$ denota o tamanho da palavra x. R é uma relação de equivalência? Justifique.

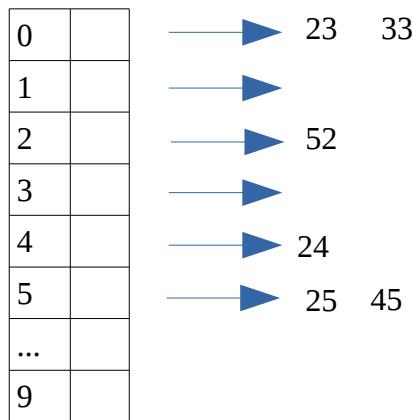
→ Fundamento conceitual de estruturas de dados conhecidas como Tabelas-Hash (Hash Tables)
Organizar dados em partições.

ID's: 23, 25, 52, 45, 24, 33, ...

Como determinar a partição que devo armazenar elemento? Função Hash

ID mod m (m é o número de partições)

Suponha m = 10



Comportamento nos casos limites:

- $m \rightarrow \infty$: evita colisões (um elemento por partição)
desperdício de espaço, mas acesso é direto (otim. para tempo)
- $m \rightarrow 1$: muitas colisões
otimiza espaço (aloca o que necessita), porém acesso totalmente sequencial
- no meio termo, compromisso entre tempo e espaço

Relações de Ordem Parcial (ROP)

- Reflexiva
- Anti-simétrica
- Transitiva

Ex: $A = \{1, 2, 3, 5, 6, 10\}$

xRy se x divide y ($y \bmod x = 0$)

$$R = \{\langle 1,1 \rangle, \langle 1,2 \rangle, \langle 1,3 \rangle, \langle 1,5 \rangle, \langle 1,6 \rangle, \langle 1,10 \rangle, \\ \langle 2,2 \rangle, \langle 2,6 \rangle, \langle 2,10 \rangle, \\ \langle 3,3 \rangle, \langle 3,6 \rangle, \\ \langle 5,5 \rangle, \langle 5,10 \rangle, \\ \langle 6,6 \rangle, \\ \langle 10,10 \rangle\}$$

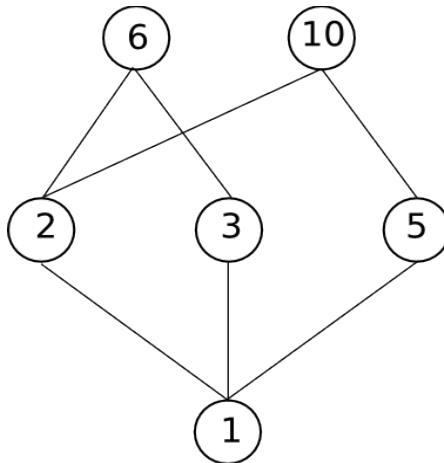
Dado um conjunto A e uma ROP em A , temos um POSET (conjunto parcialmente ordenado)

Um POSET consiste em uma tupla (A, R) em que R é uma ROP em A .

Todo POSET é representado graficamente por um diagrama de Hasse

Diagrama de Hasse

1. Desenhar cada elemento de A como um ponto de modo que se xRy , então x vem abaixo de y . Desenhar um arco entre os pontos.
2. Não há arcos para reflexidade (não há loops)
3. Não há arcos para transitividade (triângulos)



No diagrama de Hasse, existe a noção de maior e menor, pois temos um POSET.

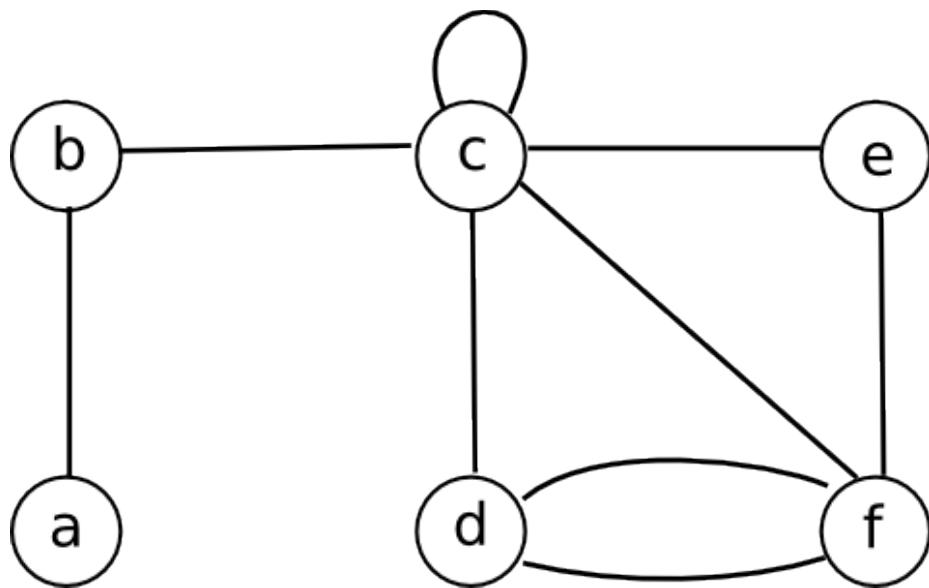
Definição: $G = (V, E)$ é um grafo se:

- i) V é um conjunto não vazio de **vértices**
- ii) $E \subseteq V \times V$ é uma relação binária qualquer no conjunto de vértices (não precisa ser apenas equivalência ou ordem parcial, pode ser qualquer coisa): conjunto de **arestas**

Obs: Convenção

Grafo não direcionado: $(a,b) = (b,a)$

Grafo direcionado ou dígrafo: $\langle a,b \rangle \neq \langle b,a \rangle$



Grafo ou Multigrafo

- Existem loops
- Existem arestas paralelas

Grafo básico simples

- Não existem loops
- Não existem arestas paralelas

Denotamos por $N(v)$ o conjunto vizinhança do vértice v . Por exemplo, $N(b) = \{a, c\}$

Definição: Grau de um vértice v : $d(v)$

É o número de vezes que um vértice v é extremidade de uma aresta. Num grafo básico simples, é o mesmo que o número de vizinhos de v . Ex: $d(a) = 1$, $d(b) = 2$, $d(c) = 6$, $d(d) = 3$, ...

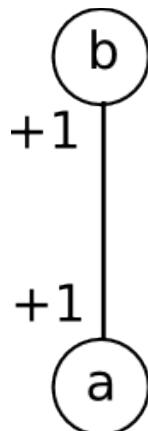
Lista de graus de G : graus dos vértices de G em ordem crescente

$$L_G = (1, 2, 2, 3, 4, 6)$$

Teorema (Handshaking Lema): A soma dos graus dos vértices de G é igual a duas vezes o número de arestas.

$$\sum_{i=1}^n d(v_i) = 2m \quad (\text{condição de existência para grafos})$$

onde $n = |V|$ e $m = |E|$ denotam respectivamente o número de vértices e arestas.



Cada aresta contribui com
+1 para o grau de cada
vértice

Propriedade: Em um grafo $G = (V, E)$ o número de vértices com grau ímpar é sempre par.
Podemos particionar V em 2 conjuntos: P (grau par) e I (grau ímpar). Assim,

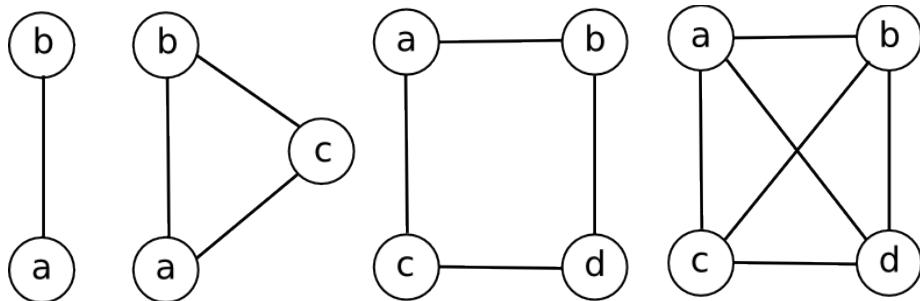
$$\sum_{i=1}^n d(v_i) = \sum_{v \in P} d(v) + \sum_{u \in I} d(u) = 2m$$

Isso implica em

$$\sum_{u \in I} d(u) = 2m - \sum_{v \in P} d(v)$$

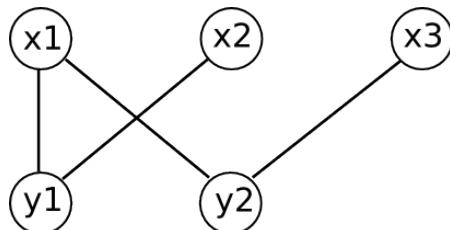
Como $2m$ é par e a soma de números pares é sempre par, resulta que a soma dos números ímpares também é par. Para que isso ocorra temos que ter $|I|$ par (número de elementos do conjunto I é par)

Def: G é k -regular $\Leftrightarrow \forall v \in V (d(v)=k)$, ou seja, $L_G = (k, k, k, k, \dots, k)$

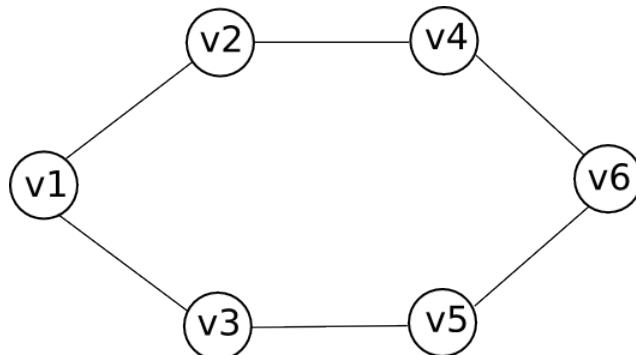


Def: Grafo Bipartido

$G = (V, E)$ é bipartido $\Leftrightarrow V = X \cup Y$ com $X \cap Y = \emptyset$ tal que
 $\forall e \in E (e = (a, b) / a \in X \wedge b \in Y)$



Ex: O grafo a seguir é bipartido ou não? Justifique sua resposta



Algoritmo para decidir se grafo é bipartido

- 1) Escolha um vértice inicial v e rotule-o como X
- 2) Para todos os vértices u ainda não rotulados e que são vizinhos a vértices rotulados como X ,

rotule-os como Y

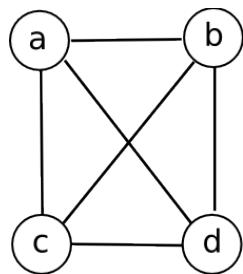
3) Para todos os vértices w ainda não rotulados que são vizinhos a vértices rotulados como Y, rotule-os como X

4) Pare quando todos os vértices do grafo estiverem rotulados.

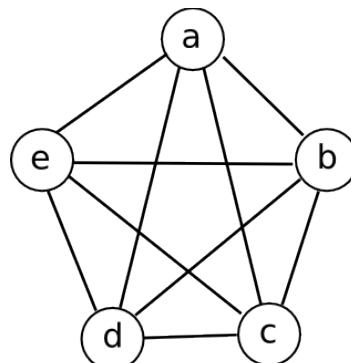
5) Se toda aresta do grafo for do tipo (X,Y), então o grafo é bipartido. Caso contrário, o grafo não é bipartido.

Def: Grafo completo

G é um grafo completo de n vértices, denotado por K_n , se cada vértice é ligado a todos os demais, ou seja, se $L_G = (n-1, n-1, n-1, \dots, n-1)$



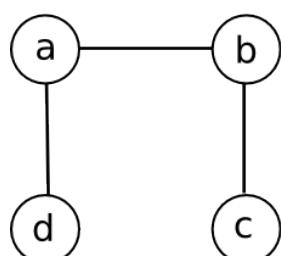
K4



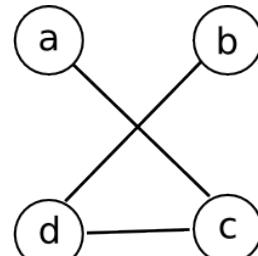
K5

O número de arestas do grafo K_n é dado por $\binom{n}{2} = \frac{n!}{(n-2)! 2!} = \frac{n(n-1)}{2}$

Def: Complementar de um grafo G: $\bar{G} = K_n - G$



G



\bar{G}

Obs: $G + \bar{G} = K_n$

Def: Subgrafo

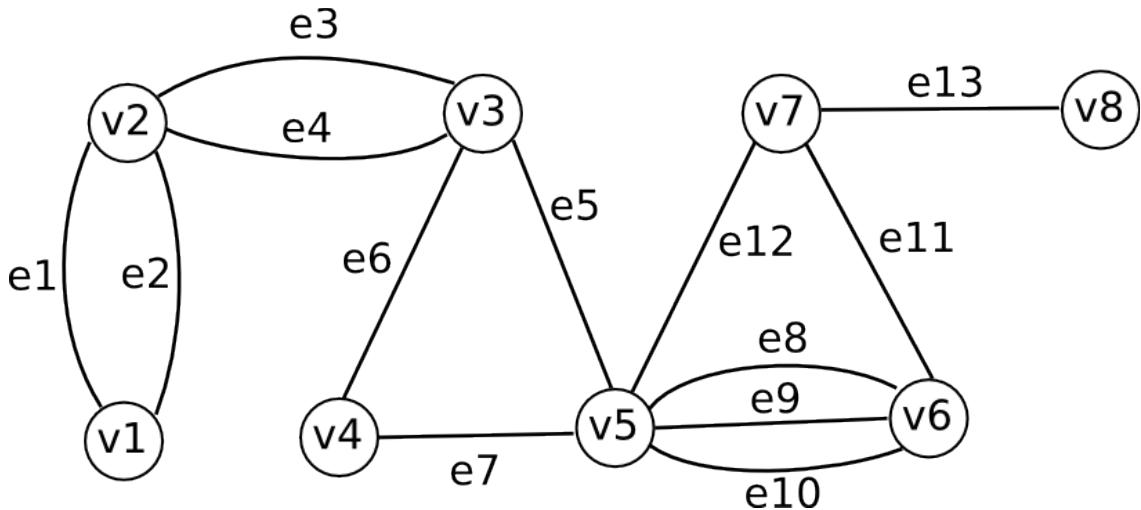
Seja $G = (V, E)$ um grafo. Dizemos que $H = (V', E')$ é um subgrafo de G se $V' \subseteq V$ e $E' \subseteq E$

Em outras palavras, é todo grafo que pode ser obtido a partir de G através de remoção de vértices e/ou arestas.

Def: Subgrafos disjuntos: não possuem vértices em comum

Subgrafos arestas disjuntos: não possuem aresta em comum

Caminhos e ciclos



a) Passeio: não tem restrição alguma quanto a vértices e arestas

$$P = v1 \ e1 \ v2 \ e1 \ v1 \ e1 \ v2$$

b) Trilha: não há repetição de arestas

$$T = v1 \ e1 \ v2 \ e3 \ v3 \ e4 \ v2$$

Se trilha é fechada, temos um circuito. Ex: $T = v1 \ e1 \ v2 \ e3 \ v3 \ e4 \ v2 \ e2 \ v1$

c) Caminho: não há repetição de vértices

$$C = v1 \ e1 \ v2 \ e3 \ v3 \ e6 \ v4 \ e7 \ v5$$

Se caminho é fechado, temos um ciclo

Obs: O comprimento/tamanho de um caminho/trilha/passeio é o número de arestas percorridas

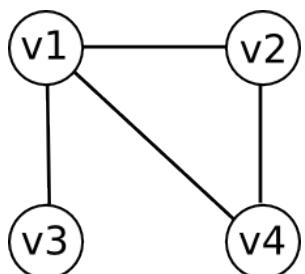
Obs: O caminho/trilha/passeio trivial é aquele composto por zero arestas

Representações computacionais de grafos

1) Matriz de adjacências A: matriz quadrada n x n definida como:

a) Grafos básicos simples

$$A_{i,j} = \begin{cases} 1, & i \leftrightarrow j \\ 0, & c.c \end{cases}$$

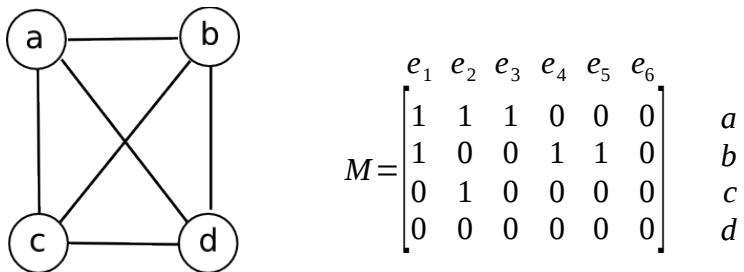


$$A = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix}$$

Propriedades básicas

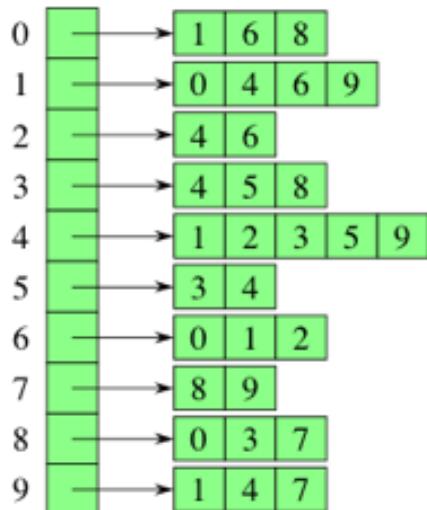
- i) $\text{diag}(A) = 0$
- ii) matriz binária
- iii) $A = A^T$ (com exceção de dígrafos)
- iv) $\sum_j A_{i,j} = d(v_i)$
- v) Esparsa
- vi) $O(n^2)$ em espaço

2) Matriz de Incidência M: matriz $n \times m$ em que as linhas referem-se aos vértices e as colunas referem-se as arestas



Obs: Loops são indicados pelo número 2 no vértice em questão

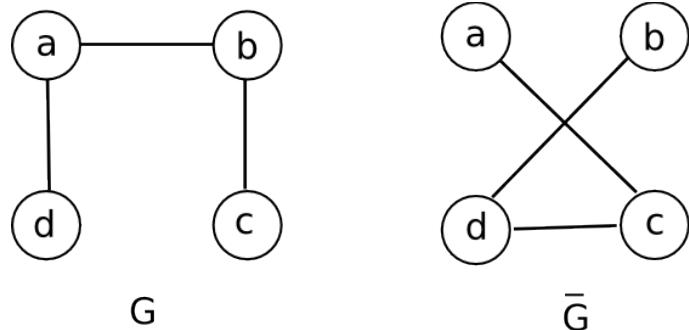
3) Lista de adjacências: estrutura do tipo hash-table (mais eficiente em termos de memória)



OBS: Em Python, na biblioteca NetworkX grafos são objetos com 2 atributos fundamentais: node e edge. Ambos são estruturas conhecidas como dicionários, e armazenam diversas informações sobre os vértices e as arestas de um grafo. É possível invocar métodos para gerar a matriz de adjacências e de incidência.

O Problema do Isomorfismo

Um problema recorrente no estudo dos grafos consiste em determinar sob quais condições 2 grafos são de fato idênticos, ou seja, queremos saber se G_1 “é igual” a G_2 . Dizemos que grafos que satisfazem essa condição de igualdade são isomorfos (iso = mesma, morfos = forma).

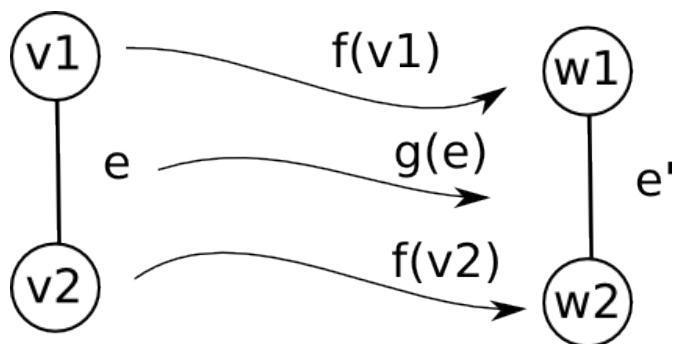


Def: $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são isomorfos se:

- i) $\exists f: V_1 \rightarrow V_2$ tal que f é bijetora (mapeamento 1 para 1)
- ii) $\exists g: E_1 \rightarrow E_2$ tal que g é bijetora

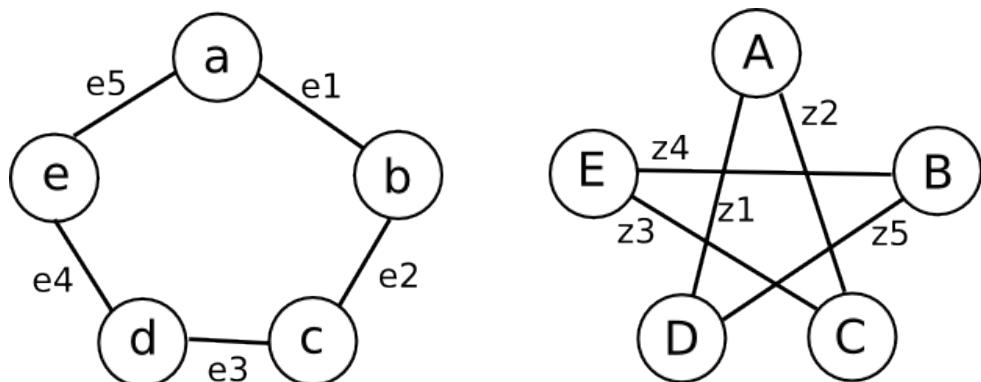
satisfazendo a seguinte restrição (*)

$$v_1 \leftarrow e \rightarrow v_2 \Leftrightarrow f(v_1) \leftarrow g(e) \rightarrow f(v_2)$$



Em termos práticos, G_1 e G_2 são isomorfos se é possível obter G_2 a partir de G_1 sem cortar arestas, ou seja, apenas movendo seus vértices (transformação isomórfica).

Exercício: Os grafos abaixo são isomorfos? Prove ou refute.



Passo 1: Encontrar mapeamento f

v	a	b	c	d	e
<hr/>					
f(v)	A	C	E	B	D

Passo 2: Encontrar mapeamento g , sujeito a restrição (*)

e	e1	e2	e3	e4	e5
<hr/>					
f(e)	z2	z3	z4	z5	z1

- i) $e1 = (a, b) \rightarrow g(e1)$ deve ser a aresta que une $f(a)$ com $f(b)$: $(A, C) = z2$
- ii) $e2 = (b, c) \rightarrow g(e2)$ deve ser a aresta que une $f(b)$ com $f(c)$: $(C, E) = z3$
- iii) $e3 = (c, d) \rightarrow g(e3)$ deve ser a aresta que une $f(c)$ com $f(d)$: $(C, E) = z4$
- iv) $e4 = (d, e) \rightarrow g(e4)$ deve ser a aresta que une $f(d)$ com $f(e)$: $(B, D) = z5$
- v) $e5 = (e, a) \rightarrow g(e5)$ deve ser a aresta que une $f(e)$ com $f(a)$: $(D, A) = z1$

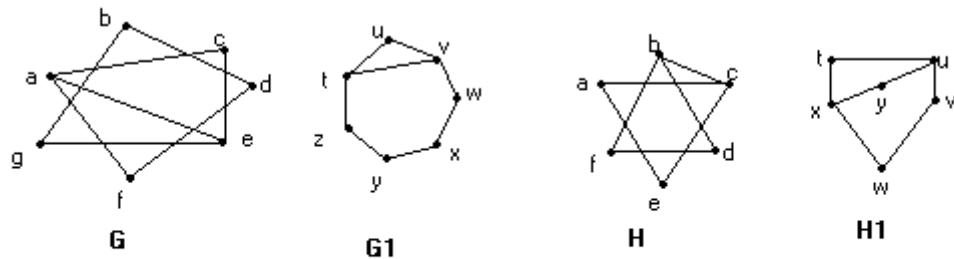
Portanto, os grafos G_1 e G_2 são isomorfos.

Propriedades Invariantes

Na determinação de isomorfismo entre grafos torna-se útil o estudo de propriedades invariantes. Em outras palavras, ao se obter medidas invariantes a transformações isomórficas, pode-se facilmente verificar que dois grafos não são isomorfos se tais medidas não forem idênticas. Sejam $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ dois grafos isomorfos. Então,

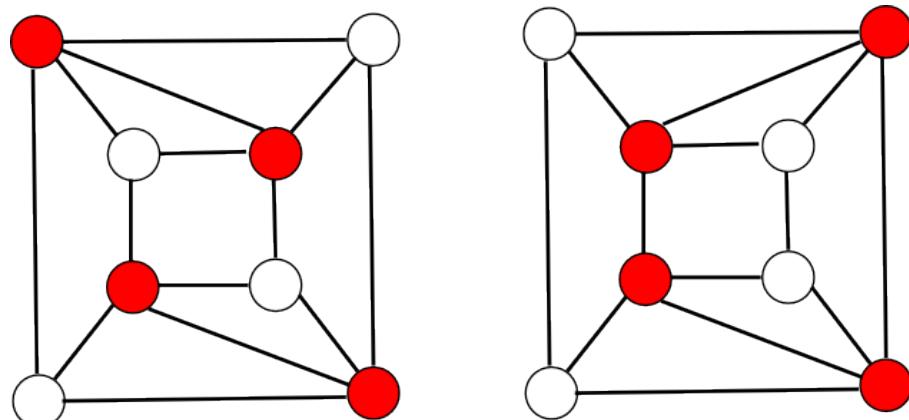
- 1) $|V_1| = |V_2|$ (o número de vértices é igual)
- 2) $|E_1| = |E_2|$ (o número de arestas é igual)
- 3) $\omega(G_1) = \omega(G_2)$ (mesmo número de componentes conexos)
- 4) $L_{G_1} = L_{G_2}$ (listas de graus são idênticas)
- 5) Ambos G_1 e G_2 admitem ciclos de comprimento k , para $k \leq n$

Ex:



Note que G é isomorfo a G_1 . Porém, H não é isomorfo a H_1 pois enquanto H admite ciclo de comprimento 3, H_1 não admite.

Ex:



Número de vértices: 8

Número de arestas: 8

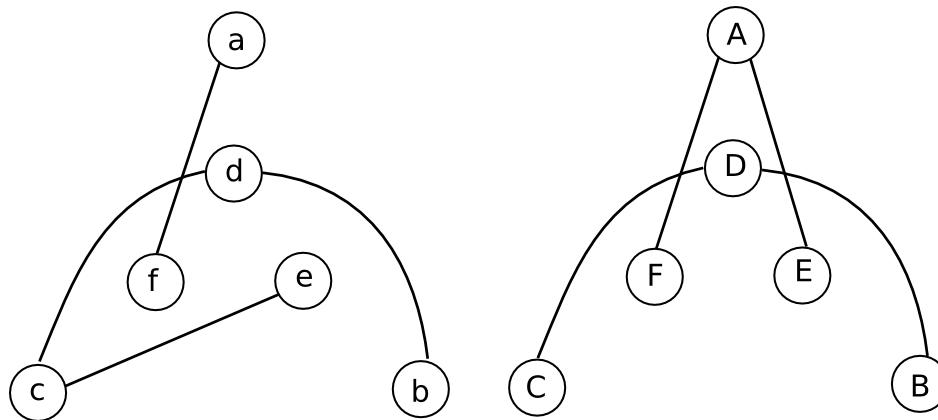
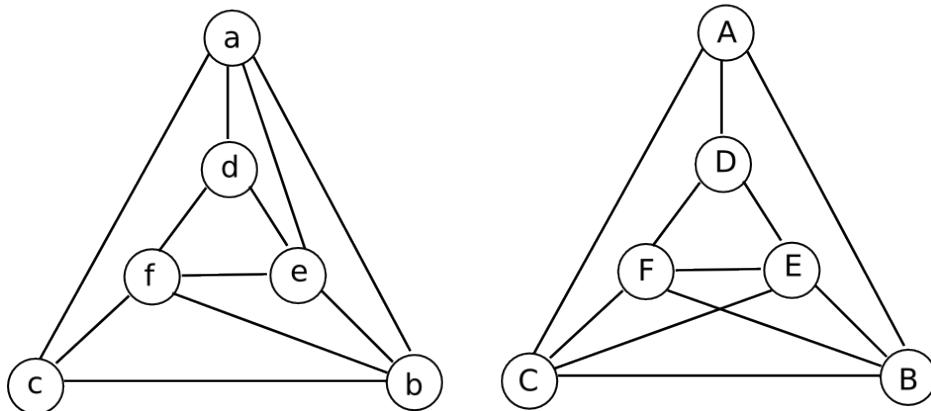
Lista de graus: (3, 3, 3, 3, 4, 4, 4, 4)

Ciclos de comprimentos 3, 4, 5, 6, 7 e 8 → OK

Apesar de não ferir nenhuma das propriedades invariantes, não podemos afirmar que os grafos são isomorfos. Na verdade, os grafos em questão não são isomorfos. Note que em um deles, todos os vértices de grau 4 (vermelho) possuem como vizinhos exatamente um outro vértice vermelho, enquanto no outro, cada vértice de grau 4 possui exatamente 2 vértices vermelhos. Isso significa uma ruptura na topologia, o que implica em corte e posterior religação de aresta.

Teorema: $G_1 \equiv G_2 \Leftrightarrow \bar{G}_1 \equiv \bar{G}_2$

Dois grafos são isomorfos se e somente se seus complementares também o forem.



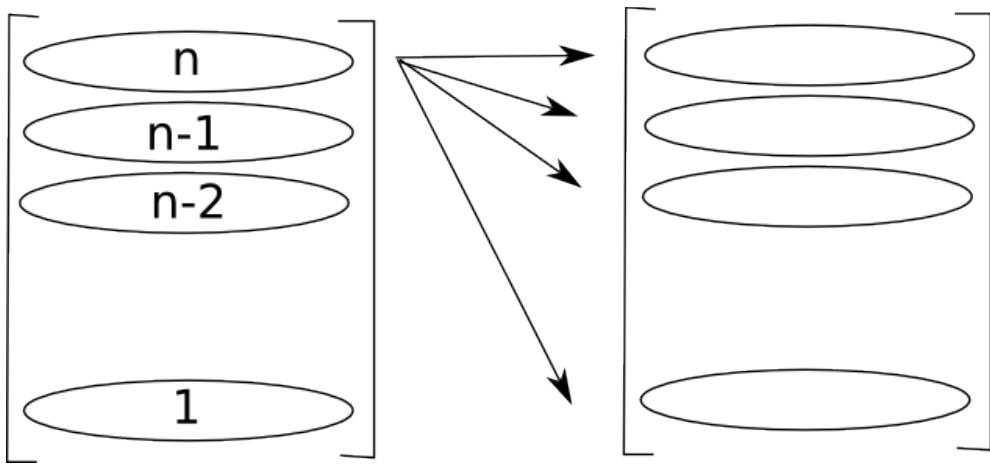
Note que no primeiro grafo os vértices de grau 2 são adjacentes, o que não ocorre no segundo.

Teorema: $G_1 \equiv G_2 \Leftrightarrow A_1 = A_2$ para alguma sequencia de permutações de linhas e colunas.

Algoritmo: Dadas duas matrizes de adjacências A1 e A2, permutar linhas e colunas de A1 de modo a chegar em A2

Complexidade:

Para as linhas temos o seguinte cenário



o que resulta num total de $n!$ possibilidades. O mesmo vale para as colunas, de forma que a complexidade do algoritmo é da ordem de $O(n!)$, tornando o método inviável para a maioria dos grafos. Atualmente, ainda não se conhecem algoritmos polinomiais para esse problema. Para algumas classes de grafos o problema é polinomial (árvores, grafos planares). Esse é o primeiro dos problemas que veremos que ainda não tem solução: o problema do isomorfismo entre grafos pode ser resolvido em tempo polinomial? Não se conhece resposta completa para a pergunta.

Coneetividade em grafos

Para fundamentar diversas propriedades de grafos é preciso apresentar aspectos relacionados a conectividade. Noções como componentes conexos e condições para conexidade são cruciais no estudo de tais objetos.

Def: Dois vértices u e v são conectados se existe um caminho P_{uv}

Def: G é conexo $\Leftrightarrow \forall u, v \in V (\exists \text{ caminho } P_{uv})$

Def: Componente conexo: subgrafo conexo máximo

Prop1: Todo grafo conexo tem $|E| \geq |V| - 1$

Prop2: Se $|E| \geq \binom{|V|-1}{2}$ então o grafo G é conexo

Prop3: Seja G é um grafo conexo. G é bipartido \Leftrightarrow Todo ciclo de G tem comprimento par

(ida) bipartido \rightarrow apenas ciclos pares

Se G é bipartido então toda aresta é (X, Y) . Se saio do lado X , único modo de voltar é passar por duas arestas, uma de X para Y e outra de Y para X . Isso implica que o comprimento de qualquer ciclo será múltiplo de 2.

Parêntesis:

p	q	!p	!q	p \rightarrow q	!q \rightarrow !p
V	V	F	F	V	V
V	F	F	V	F	
F	V	V	F	V	V
F	F	V	V	V	

(volta) apenas ciclos pares \rightarrow G bipartido = G não bipartido \rightarrow existe ciclos ímpares

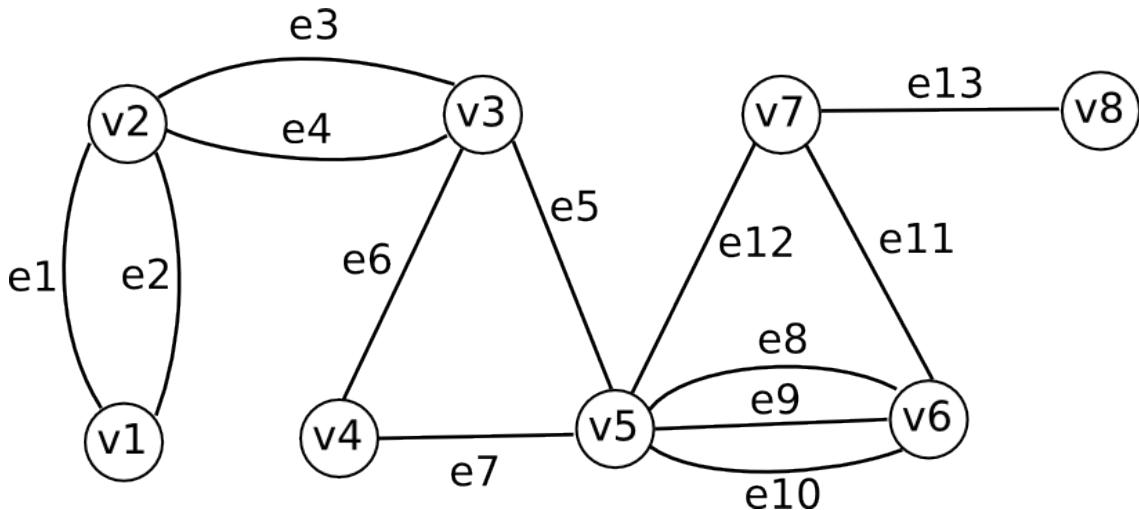
Se G não é bipartido, existe ao menos uma aresta (X, X) ou (Y, Y), o que implica na existência de um triângulo em G (ciclo de comprimento 3).

Def: A distância geodésica entre u e v, denotada por $d(u,v)$ é o comprimento do menor caminho entre u e v

Métricas

a) Excentricidade de um vértice: $e(v) = \max\{d(v,u) \mid u, v \in V \wedge u \neq v\}$

Em palavras, é o quanto longe posso chegar a partir do vértice v, andando apenas por caminhos mínimos (é a máxima distância geodésica de v a qualquer outro u). Maior menor caminho.



$$\begin{array}{llll} e(v_1) = 5 & e(v_2) = 4 & e(v_3) = 3 & e(v_4) = 3 \\ e(v_5) = 3 & e(v_6) = 4 & e(v_7) = 4 & e(v_8) = 5 \end{array}$$

b) Raio de um grafo: $r(G) = \min\{e(v) \mid v \in V\}$

É a mínima excentricidade de um vértice

$$r(G) = 3$$

c) Diâmetro de um grafo: $d(G) = \max\{e(v) \mid v \in V\}$

É a máxima excentricidade de um vértice

$$d(G) = 5$$

Obs: Diâmetro da internet e redes sociais (6 graus de separação)

Prop: $r(G) \leq d(G) \leq 2r(G)$

Pode ser verificado observando os casos extremos

i) o grafo mais compacto que existe: K_n (excentricidades dos vértices menor possível)

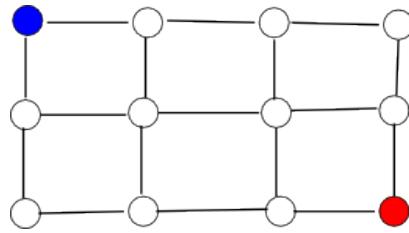
ii) o grafo mais espalhado que existe (grafo caminho): P_n

Número de passeios

Em diversas ocasiões precisamos saber de quantas formas podemos nos locomover de um ponto a outro de um grafo (exemplo numa matriz que é mais fácil de ver)

Para sair do ponto azul e chegar no ponto vermelho há várias possibilidades. Alguns exemplos são E, E, E, B, B ou E, E, B, B, E ou E, E, B, E, B etc...

É fácil enumerar todas elas pois a grade retangular é um reticulado e possui um padrão bem definido. Mas e no caso de um grafo G qualquer?



Teorema: Seja A a matriz de adjacência de G e $A^k = A \cdot A \cdot A \dots \cdot A$. O elemento $a_{ij}^{(k)}$ da matriz A^k denota o número de passeios de tamanho k que existem entre v_i e v_j

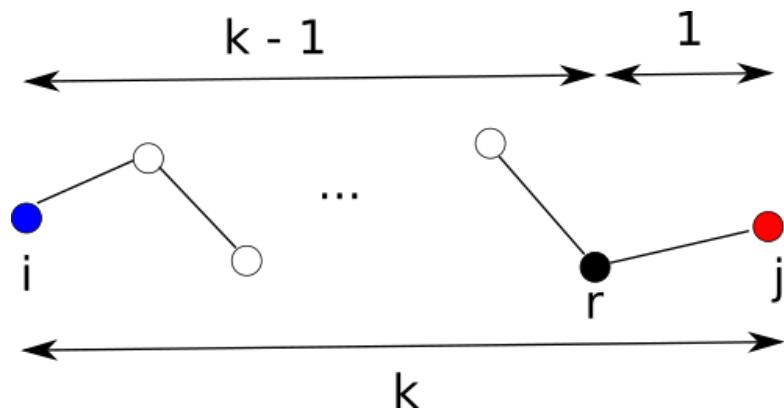
PROVA:

A prova é por indução em k (comprimento do passeio)

Passo 1: Para $k = 1$, trivialmente satisfeito pois $a_{ij}^{(1)}$ é o número de arestas entre v_i e v_j (base)

Passo 2: Assumir $k > 1$ e supor que afirmação é válida para $k - 1$. Mostrar que afirmação vale para k . (passo de indução)

i) Considere um passeio de tamanho k entre v_i e v_j



Note que tal passeio pode ser decomposto em 2 partes: um passeio de tamanho $k - 1$ de v_i a um vértice v_r vizinho a v_j e a aresta $v_r - v_j$

ii) Pela nossa suposição ($a_{ij}^{(k-1)}$ denota o número de passeios de tamanho $k - 1$ entre v_i e v_j), o número de possíveis passeios de tamanho $k - 1$ de v_i a um vizinho v_r de v_j é $a_{ir}^{(k-1)}$ (de A^{k-1}) e o número de passeios de tamanho 1 de v_r a v_j é a_{rj} (de A). Então o número de passeios de tamanho k de v_i a v_j via v_r (porta de entrada é v_r) é $a_{ir}^{(k-1)} a_{rj}$

iii) O número total de passeios de tamanho k entre v_i e v_j pode ser obtido computando o termo anterior para todas as possíveis portas de entrada em v_j , ou seja:

$$a_{i1}^{(k-1)} a_{1j} + a_{i2}^{(k-1)} a_{2j} + a_{i3}^{(k-1)} a_{3j} + \dots + a_{ir}^{(k-1)} a_{rj} + \dots + a_{in}^{(k-1)} a_{nj}$$

Note que se um vértice w não é porta de entrada para v_j então então $a_{wj}=0$ o que automaticamente anula o valor da parcela.

A equação anterior é justamente o elemento a_{ij} da matriz A^k pois

$$\begin{array}{c} \text{coluna j} \\ \left[\begin{array}{c} \text{linha i} \\ a_{i1}^{(k-1)} \ a_{i2}^{(k-1)} \ \dots \ a_{in}^{(k-1)} \end{array} \right] \left[\begin{array}{c} a_{1j} \\ a_{2j} \\ \vdots \\ a_{nj} \end{array} \right] = \left[\begin{array}{c} \dots \ a_{ij}^{(k)} \ \dots \\ \vdots \\ \dots \end{array} \right] \end{array} \\ A^{(k-1)} \qquad \qquad \qquad A \qquad \qquad \qquad A^{(k)}$$

Então, como a partir da suposição de que a afirmação é válida para $k - 1$, verificamos sua validade para k , a prova por indução está completa.

Cadeias de Markov e Random Walks

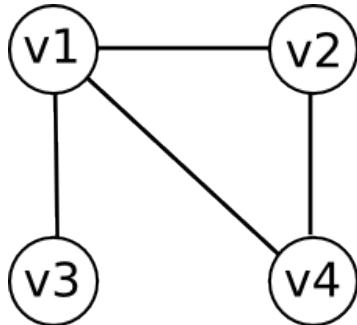
Random Walks

Cenário: Andar do bêbado

Seja $G = (V, E)$ um grafo básico simples com matriz de adjacência A . Então, podemos definir a matriz P como segue:

$$P = \Delta^{-1} A \quad \text{onde} \quad \Delta^{-1} = \begin{bmatrix} \frac{1}{d(v_1)} & 0 & 0 & 0 \\ 0 & \frac{1}{d(v_2)} & 0 & 0 \\ \cdot & \cdot & \cdot & \cdot \\ 0 & 0 & 0 & \frac{1}{d(v_n)} \end{bmatrix}$$

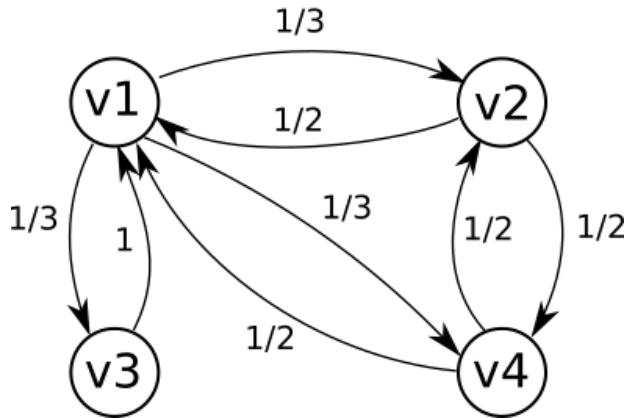
Ex:



$$P = \Delta^{-1} A = \begin{bmatrix} 1/3 & 0 & 0 & 0 \\ 0 & 1/2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1/2 \end{bmatrix} \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/2 & 0 & 0 & 1/2 \\ 1 & 0 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \end{bmatrix}$$

Note que P não é simétrica! (Elas não definem um grafo mas sim um dígrafo)

Diagrama de estados de P



Note que isso significa que, para todo i, j temos:

$$p_{ij} = \frac{a_{ij}}{\sum_j a_{ij}} = \frac{a_{ij}}{d(v_i)}$$

ou seja, $p_{ij} \in [0, 1]$ (representa a probabilidade de sair de i e chegar em j com uma única aresta)

Chamamos P de matriz de probabilidades de transição de estados, pois ela define um processo aleatório conhecido como Cadeia de Markov. Toda matriz P pode ser representada graficamente por um diagrama de estados.

Cadeias de Markov Homogêneas

Def: Uma cadeia de Markov homogênea de estados finitos e tempo discreto pode ser definida pela tupla:

$$CM = (S, X_k, P, \vec{w}^{(k)}) \quad \text{onde}$$

- i) $S = \{s_1, s_2, \dots, s_n\}$ é o conjunto de estados
- ii) X_k é uma variável aleatória que assume valores em S
- iii) $\vec{w}^{(k)}$ é um vetor de probabilidades de cada estado no tempo k.

$$\vec{w}^{(k)} = [p(X_k=s_0), p(X_k=s_1), p(X_k=s_2), p(X_k=s_3), \dots, p(X_k=s_n)]$$

o i-ésimo elemento de $\vec{w}^{(k)}$ denota a probabilidade de no tempo k estarmos no estado s_i

$\vec{w}^{(0)}$: probabilidade de iniciar o processo em cada estado: na analogia com autômatos finitos ao invés de um único estado inicial, podemos ter vários mas com diferentes probabilidades

Note que sempre devemos ter $\sum_i w_i = 1$

- iv) P é a matriz de probabilidades de transição de estados (função de transição do autômato)

CM homogênea significa que a matriz P não muda no tempo.

Porque uma matriz?

Propriedade Markoviana (cadeia de Markov): o futuro só depende do presente e não do passado anterior

Para uma sequência $X_0, X_1, X_2, \dots, X_n$ de observações a probabilidade conjunta é dada por:

$$\begin{aligned} P(X_0, X_1, X_2, \dots, X_n) &= \prod_{t=0}^T P(X_t | X_0, X_1, \dots, X_{t-1}) = \\ &= P(X_0) P(X_1 | X_0) P(X_2 | X_0, X_1) P(X_3 | X_0, X_1, X_2) \dots P(X_T | X_0, X_1, X_2, \dots, X_{T-1}) \end{aligned}$$

Se processo é Markoviano (sem memória) então:

$$P(X_T | X_0, X_1, X_2, \dots, X_{T-1}) = P(X_T | X_{T-1})$$

A probabilidade de estar no estado T dado que passei por 0, 1, 2, ... T-1 só depende do último estado em que estive: T-1 (cadeias de Markov de primeira ordem podem ser representadas por matrizes de transição em que a probabilidade de transicionar de i para j é dado por $P_{ij} = P(X_j | X_i)$)

Em outras palavras, a probabilidade de eu acessar um estado s3 no tempo t não depende do histórico todo mas apenas de onde eu estava no tempo anterior

Equações de Chapman-Kolmogorov

Na forma vetorial, a relação entre distribuição de probabilidades dos estados no tempo k e k-1 pode ser expressa por uma equação linear, dada por:

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} P$$

Ex: Matriz de Oz

$$P = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{bmatrix} \quad \begin{matrix} R \\ C \\ S \end{matrix} \quad \text{3 estados: R (rain), C (cloud), S (sun)}$$

Supor $\vec{w}^{(0)} = [0, 0, 1]$ (começa com dia de sol), o que vai acontecer no longo prazo, ou seja, quem será $\vec{w}^{(k)}$ para $k \rightarrow \infty$. Distribuição converge ou diverge? Depende diretamente de propriedades matemáticas relacionadas as componentes do modelo.

Def: Uma CM homogênea é irredutível se é possível atingir qualquer estado i a partir de qualquer outro j.

Def: Uma CM homogênea é aperiódica se $\exists k | P^k$ contém apenas elementos não nulos

Def: Se um CM homogênea é irredutível e aperiódica então ela é ergódica

Pode-se mostrar que caminhadas aleatórias em grafos não direcionados satisfazem:

- i) P é irreduzível $\Leftrightarrow G$ é conexo
- ii) P é aperiódica $\Leftrightarrow G$ não é bipartido
- iii) P é reversível, isto é atinge o equilíbrio (no equilíbrio a matriz P se comporta como simétrica)

Teorema Fundamental das CM's

Seja P a matriz de transição de uma CM homogêna ergódica. Então a distribuição estacionária existe, é única e:

$$\lim_{k \rightarrow \infty} P^k \quad \text{com} \quad W = \begin{bmatrix} \vec{w}^{(k)} \\ \dots \\ \vec{w}^{(k)} \end{bmatrix} \quad \text{onde} \quad \vec{w}^{(k)} \quad \text{é a distribuição estacionária}$$

Obs: Note que $\vec{w}^{(k)}$ não depende de $\vec{w}^{(0)}$ (de onde começo).

Power Method

Método iterativo para obter a distribuição de probabilidades dos estados num tempo k .

$$\vec{w}^{(k)} = \vec{w}^{(k-1)} P = (\vec{w}^{(k-2)} P) P = ((\vec{w}^{(k-3)} P) P) P = \dots$$

$$\vec{w}^{(k)} = \vec{w}^{(0)} P^k$$

Solução analítica

Pode-se mostrar que no caso de CM's ergódicas é possível computar a distribuição estacionária teórica (analítica). Por exemplo, no caso de caminhadas aleatórias em grafos não direcionados, conexos e não bipartidos podemos obter $\vec{w}^{(k)}$ sem adotar o método iterativo.

No equilíbrio temos (pois a CMH é reversível - função de transição se comporta como se fosse simétrica):

$$w_i p_{i,j} = w_j p_{j,i} \quad (\text{prob. de estar em } i \text{ e trans. para } j = \text{prob. de estar em } j \text{ e trans. de } j \text{ para } i)$$

Da matriz P sabemos que $p_{i,j} = \frac{1}{d(v_i)}$ então:

$$w_i \frac{1}{d(v_i)} = w_j \frac{1}{d(v_i)} = K$$

Assim, temos:

$$w_i = K d(v_i) \quad (*)$$

Somando ambos os lados em relação a i :

$$\sum_{i=1}^n w_i = K \sum_{i=1}^n d(v_i)$$

Pelo Handshaking Lema, a equação anterior fica:

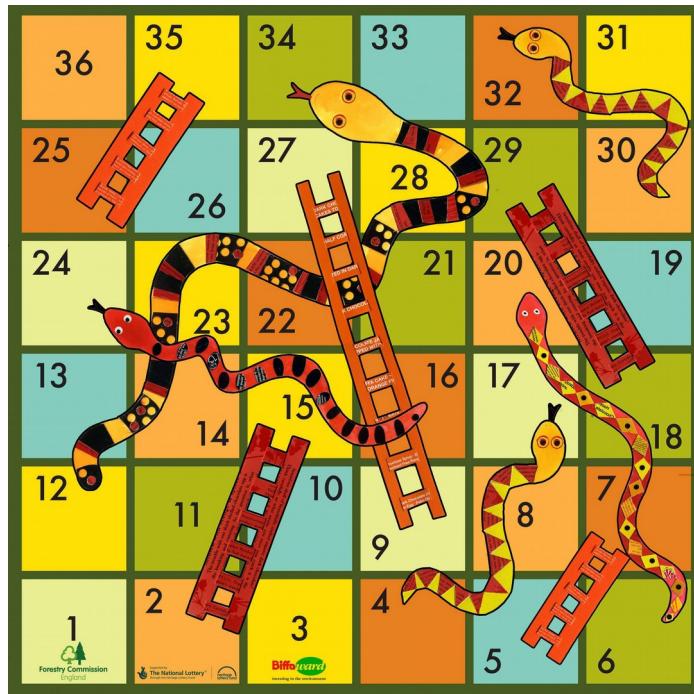
$$1 = K 2|E| \quad \text{ou seja,} \quad K = \frac{1}{2|E|}$$

Portanto, de (*) temos que a distribuição estacionária é dada por:

$$\vec{w} = \left[\frac{d(v_1)}{2|E|}, \frac{d(v_2)}{2|E|}, \dots, \frac{d(v_n)}{2|E|} \right]$$

onde $|E|$ é o número de arestas.

Snakes and Ladders



Snakes and Ladders é um jogo de tabuleiro em que a cada rodada um jogador joga uma moeda não viciada e avança 1 casa se obtiver cara ou avança 2 casas se obtiver coroa. Se o jogador para no pé da escada, então ele imediatamente sobe para o topo da escada. Se o jogador cai na boca de um cobra então ele imediatamente escorrega para o rabo. O jogador sempre inicia no quadrado de número 1. O jogo termina quando ele atinge o quadrado de número 36. Com base nas informações, responda:

- a)** Especifique o diagrama de estados da cadeia de Markov que representa o jogo, computando para isso a matriz de transição de estados P . O que podemos dizer sobre o estado 36?
- b)** Implemente um programa/script para calcular a distribuição estacionária da cadeia de Markov homogênea em questão. Qual é a probabilidade de um jogador vencer o jogo, ou seja, qual a probabilidade de se atingir o estado 36 no longo prazo? Considere $k = 50$ um número suficiente de iterações no Power Method. Utilize outros vetores iniciais e observe o comportamento do método.

O Modelo Pagerank

Motivação: Caminhadas aleatórias em dígrafos (i.e., internet)

Problema: CM não é irredutível nem aperiódica, distribuição estacionária $\vec{w}^{(k)}$ não é única

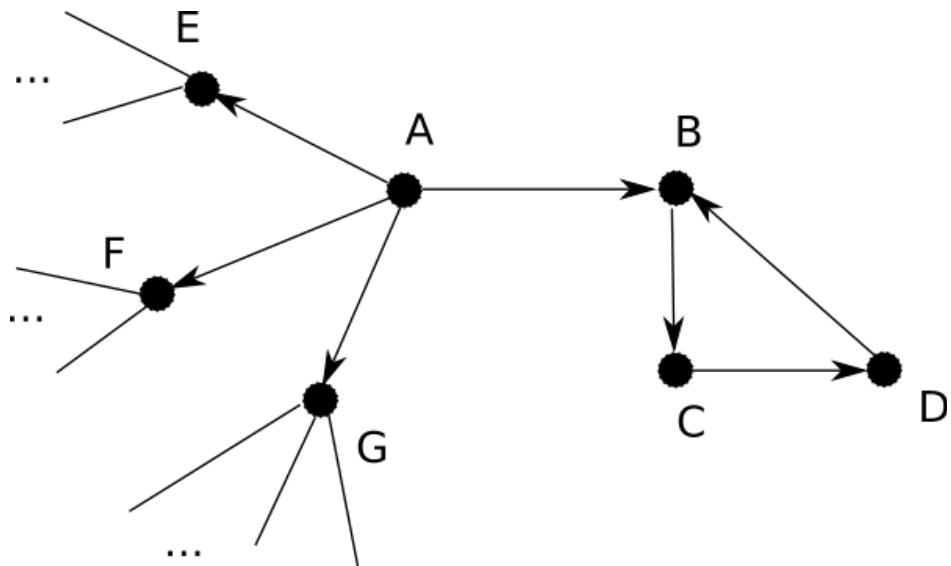
(depende diretamente de onde começo: $\vec{w}^{(0)}$)

É preciso ajustar o modelo padrão para resolver problemas:

a) Presença de *dangling nodes* (pontos sem saída): estados absorventes, uma vez acessado nunca mais sairá.

Solução: permitir eventuais saltos, com pequena probabilidade
Ligar todos os nós com todos com probabilidade uniforme ($1/n$)

b) Pode não existir k tal que seja possível estar em qualquer estado com probabilidade não nula (ficamos presos em um subgrafo, o que torna a caminhada periódica)



Se inicio em A em $t = 0$ e vou para B em $t = 1$, caminhada fica periódica (BCDBCDBCD...)

Solução: permitir loops (posso permanecer num estado com uma dada probabilidade)
Introduzir elementos não nulos na diagonal de P

Ideia geral: processo estocástico que modela um navegador que
- realiza uma caminhada aleatória padrão com probabilidade $(1-\alpha)$
- salta para um estado aleatório com probabilidade α (teleporte)

Pode-se mostrar que esse processo é equivalente a modelar uma CM caracterizada pela seguinte matriz de transição

$$\bar{P} = (1-\alpha)P + \alpha \frac{1}{n}U \quad (\text{Google matrix}) \text{ onde}$$

$$U = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \dots & \dots & \dots & \dots \\ 1 & 1 & \dots & 1 \end{bmatrix} \quad \text{denota uma matriz } n \times n \text{ de 1's e } \alpha \in [0,1]$$

Resumo

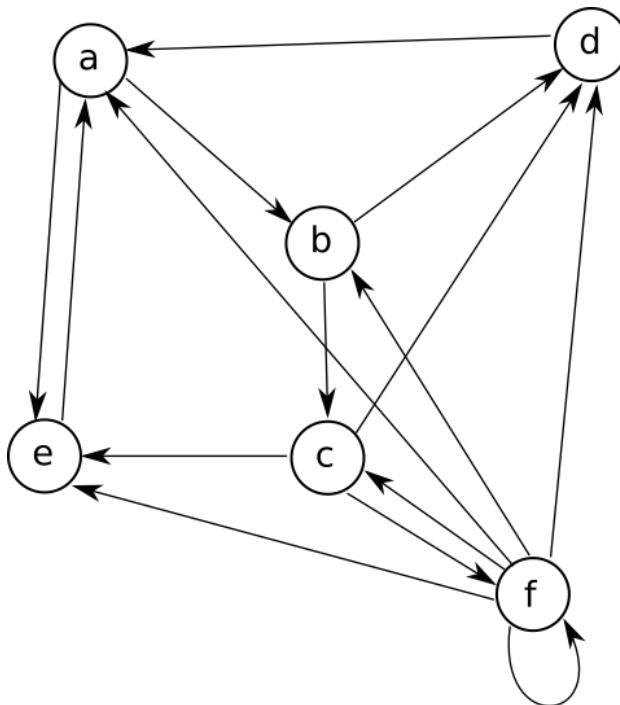
G (digrafo) \rightarrow A (mat. adj.) \rightarrow P (probs. RW padrão) \rightarrow \bar{P} (escolhido α)

Observações:

1. $\alpha=0 \rightarrow \bar{P}=P$ (RW padrão)
Distribuição estacionária depende de $\vec{w}^{(0)}$ (não é única)
2. $\alpha=1 \rightarrow$ distribuição estacionária totalmente não informativa (descarta completamente a topologia de G)

Objetivo: encontrar compromisso entre os casos limites

Ex:



De posse do grafo, geramos a matriz de transição de probabilidade P.

$$P = \begin{bmatrix} 0 & 1/2 & 0 & 0 & 1/2 & 0 \\ 0 & 0 & 1/2 & 1/2 & 0 & 0 \\ 0 & 0 & 0 & 1/3 & 1/3 & 1/3 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{bmatrix}$$

Obs: Se $\exists v \in V | d_{\text{(out)}}=0$ (dangling node)

$$P = P + \frac{1}{n} \vec{z} \vec{u}^T \quad (\text{na prática preenche a linha de zeros com } 1/n)$$

onde $\vec{u}^T = [1, 1, 1, \dots, 1]$ (vetor de 1's) e $\vec{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \dots \\ 0 \end{bmatrix}$ (indicador de quais linhas são nulas)

Em resumo, as linhas de P onde todos os elementos são nulos, terão valor igual a $1/n$

Vamos considerar um valor de $\alpha=0.15$.

$$\bar{P} = 0.85 P + 0.15 \frac{1}{6} U$$

$$\text{Linha 1: } \frac{85}{100} \frac{1}{2} + \frac{15}{100} \frac{1}{6} = \frac{85}{200} + \frac{5}{200} = \frac{90}{200} = \frac{18}{40} \quad \text{e} \quad \frac{15}{100} \frac{1}{6} = \frac{5}{200} = \frac{1}{40}$$

$$\text{Linha 2: } \frac{85}{100} \frac{1}{2} + \frac{15}{100} \frac{1}{6} = \frac{85}{200} + \frac{5}{200} = \frac{90}{200} = \frac{18}{40} \quad \text{e} \quad \frac{15}{100} \frac{1}{6} = \frac{5}{200} = \frac{1}{40}$$

$$\text{Linha 3: } \frac{85}{100} \frac{1}{3} + \frac{15}{100} \frac{1}{6} = \frac{85}{300} + \frac{5}{200} = \frac{185}{600} = \frac{37}{120} \quad \text{e} \quad \frac{15}{100} \frac{1}{6} = \frac{5}{200} = \frac{1}{40} = \frac{3}{120}$$

$$\text{Linha 4: } \frac{85}{100} + \frac{15}{100} \frac{1}{6} = \frac{85}{100} + \frac{5}{200} = \frac{175}{200} = \frac{35}{40} \quad \text{e} \quad \frac{15}{100} \frac{1}{6} = \frac{5}{200} = \frac{1}{40}$$

$$\text{Linha 5: } \frac{85}{100} + \frac{15}{100} \frac{1}{6} = \frac{85}{100} + \frac{5}{200} = \frac{175}{200} = \frac{35}{40} \quad \text{e} \quad \frac{15}{100} \frac{1}{6} = \frac{5}{200} = \frac{1}{40}$$

$$\text{Linha 6: } \frac{85}{100} \frac{1}{6} + \frac{15}{100} \frac{1}{6} = \frac{85}{600} + \frac{15}{100} = \frac{100}{600} = \frac{1}{6}$$

$$P = \begin{bmatrix} 1/40 & 18/40 & 1/40 & 1/40 & 18/40 & 1/40 \\ 1/40 & 1/40 & 18/40 & 18/40 & 1/40 & 1/40 \\ 3/120 & 3/120 & 3/120 & 37/120 & 37/120 & 37/120 \\ 35/40 & 1/40 & 1/40 & 1/40 & 1/40 & 1/40 \\ 35/40 & 1/40 & 1/40 & 1/40 & 1/40 & 1/40 \\ 1/6 & 1/6 & 1/6 & 1/6 & 1/6 & 1/6 \end{bmatrix}$$

Note que todas as linhas somam 1.

Essa matriz modela uma CM irredutível e aperiódica, de modo que agora a distribuição estacionária é única e não depende de $\vec{w}^{(0)}$. Na prática isso significa que o ranking de pageranks é único. Seria um problema para o Google por exemplo se os rankings das páginas com maiores pageranks fosse dependente da inicialização. Cada dia teríamos um diferente, inconsistência seria grande.

Para se chegar na distribuição estacionária, basta aplicar o Power Method: $\vec{w}^{(k)} = \vec{w}^{(k-1)} \bar{P}$

Pergunta: Existe solução analítica ou preciso usar o Power method?

No equilíbrio, isto é, quando a distribuição estacionária converge temos:

$$\vec{w} = \vec{w} \bar{P}$$

$$\vec{w} = \vec{w} \left[(1 - \alpha) P + \alpha \frac{1}{n} U \right]$$

$$\vec{w} = (1 - \alpha) \vec{w} P + \alpha \frac{1}{n} \vec{w} U$$

Como $\vec{w} U = [1, 1, 1, \dots, 1] = \vec{u}$ e sabendo que I denota a matriz identidade temos:

$$I\vec{w} = (1-\alpha)\vec{w}P + \alpha \frac{1}{n}\vec{u}$$

$$I\vec{w} - (1-\alpha)\vec{w}P = \alpha \frac{\vec{u}}{n}$$

$$I\vec{w} - (1-\alpha)\vec{w}P = \alpha \frac{\vec{u}}{n}$$

o que finalmente nos leva a:

$$\vec{w} = \alpha \frac{\vec{u}}{n} [I - (1-\alpha)P]^{-1}$$

Assim, embora exista uma solução analítica ela é inviável devido a necessidade de inversão da matriz. Imagine inverter uma matriz referente a internet toda! É computacionalmente inviável.

Interpretação do Pagerank

Pode-se reescrever \vec{w} de modo a expressar a soma de uma P.G. infinita. Note que

$$S_\infty = \frac{1}{1-q} = (1-q)^{-1} \quad \text{onde } q \text{ é a razão da progressão geométrica}$$

Identificando a razão como sendo $(1-\alpha)P$ temos:

$$[I - (1-\alpha)P]^{-1} = \sum_{k=1}^{\infty} [(1-\alpha)P]^k$$

o que nos leva a:

$$\vec{w} = \alpha \frac{\vec{u}}{n} \sum_{k=1}^{\infty} (1-\alpha)^k P^k = \frac{\vec{u}}{n} [(1-\alpha)P + (1-\alpha)^2 PP + (1-\alpha)^3 PPP + \dots] \alpha$$

$\frac{\vec{u}}{n}$ probabilidade uniforme de escolher uma página aleatória

$[(1-\alpha)P + (1-\alpha)^2 PP + (1-\alpha)^3 PPP + \dots]$ probabilidade de navegar com 1, 2, 3, ..., K passos

α probabilidade de saltar/parar

Essa expressão fornece a interpretação conhecida como “The Impatient Surfer”, onde os coeficientes w_i representam as probabilidades de um usuário qualquer tendo iniciado sua navegação em uma página arbitrária, parar exatamente na página i.

Convergência: quanto rápido \vec{w} se aproxima da distribuição estacionária?

Pode-se mostrar que a taxa de convergência para \vec{w} depende de λ_2 , ou seja, o segundo maior autovalor da matriz \bar{P} . Quanto maior λ_2 melhor. Além disso, pode-se verificar que

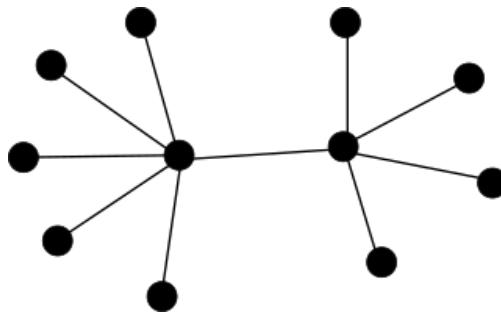
$$|\lambda_2| \leq (1-\alpha)$$

Em geral, temos $\alpha=0.1$, $\alpha=0.15$

Árvores

Árvores são grafos especiais com diversas propriedades únicas. Devido a essas propriedades são extremamente importantes na resolução de vários tipos de problemas práticos. Veremos ao longo do curso que vários problemas que estudaremos se resumem a: dado um grafo G , extrair uma árvore T a partir de G , de modo que T satisfaça uma certa propriedade (como por exemplo, mínima profundidade, máxima profundidade, mínimo peso, mínimos caminhos, etc).

Def: Um grafo $G = (V, E)$ é uma árvore se G é acíclico e conexo.

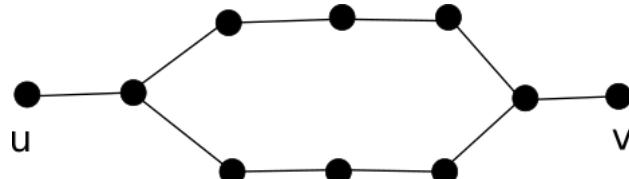


Propriedades

Prop1: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

(ida) $p \rightarrow q = !q \rightarrow !p$
 \nexists um único caminho entre quaisquer $u, v \rightarrow G$ não é uma árvore

- a) Pode existir um par u, v tal que \nexists caminho (zero caminhos). Isso implica em G desconexo, o que implica que G não é uma árvore
- b) Pode existir um par u, v tal que \exists mais de um caminho.

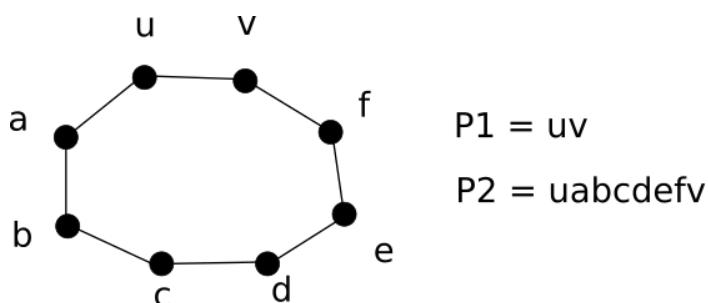


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

(volta) $q \rightarrow p = !p \rightarrow !q$

G não é árvore $\rightarrow \nexists$ único caminho entre quaisquer u, v

Para G não ser árvore, G deve ser desconexo ou conter um ciclo. Note que no primeiro caso existe um par u, v tal que não há caminho entre eles. Note que no segundo caso existem 2 caminhos entre u e v , conforme ilustra a figura

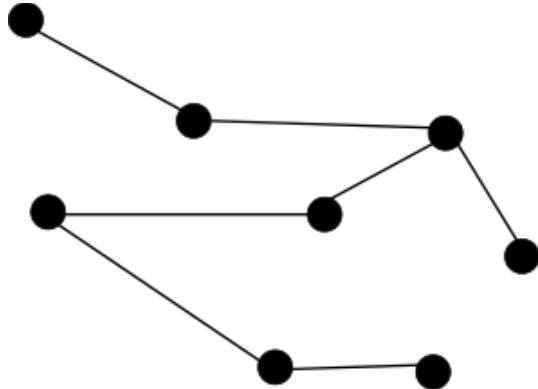


Prop2: Se $G = (V, E)$ é uma árvore com $|V| = n$ então $|E| = n - 1$

Como uma árvore é um grafo acíclico e conexo, deve ter exatamente $n - 1$ arestas pois caso contrário temos as seguintes contradições:

- 1) se $|E| < n - 1$, G seria desconexo (não é árvore);
- 2) se $|E| > n - 1$, G conteria um ciclo (não é árvore);

Árvore é um grafo T tal que a adição de qualquer aresta e gera um ciclo em $T + e$

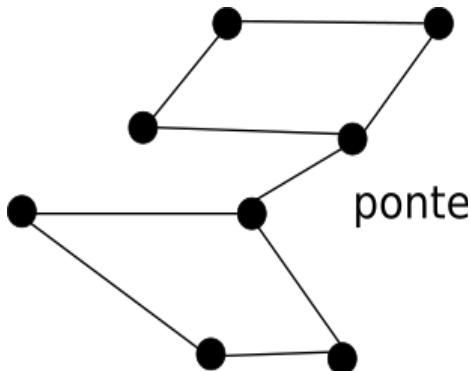


Prop3: Toda árvore com mais de 1 vértice possui ao menos 2 folhas

Considere dois vértices quaisquer u, v na árvore. Então, existe um caminho P entre eles. É possível estender P ? Caso sim, escolha uma aresta incidente a u ou v e defina um novo caminho estendido P' . Caso não seja possível estender, significa que os vértices extremidades do caminho possuem grau 1 e portanto são folhas.

Def: Uma aresta $e \in E$ é ponte se $G - e$ é desconexo

Ou seja, a remoção de uma aresta ponte desconecta o grafo

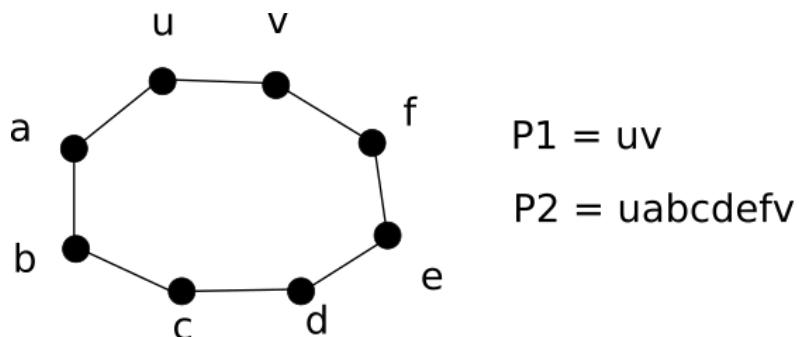


Como identificar arestas ponte?

Prop: Uma aresta $e \in E$ é ponte \Leftrightarrow aresta não pertence a um ciclo C

(ida) $p \rightarrow q = !q \rightarrow !p$

$e \in C \rightarrow$ aresta não é ponte



Como aresta pertence a um ciclo C , há 2 caminhos entre u e v . Logo a remoção da aresta $e = (u,v)$ não impede que o grafo seja conexo, ou seja, $G - e$ ainda é conexo. Portanto, e não é ponte

(volta) $q \rightarrow p = !p \rightarrow !q$
aresta não é ponte $\rightarrow e \in C$

Se aresta não é ponte então $G - e$ ainda é conexo. Se isso ocorre, deve-se ao fato de que em $G - e$ ainda existe um caminho entre u e v que não passa por e . Logo, em G existem existem 2 caminhos, o que nos leva a conclusão de que a união entre os 2 caminhos gera um ciclo C .

Prop4: G é uma árvore \Leftrightarrow Toda aresta é ponte

(ida) $p \rightarrow q = !q \rightarrow !p$
 \exists aresta não ponte $\rightarrow G$ não é árvore

A existência de uma aresta não ponte implica na existência de ciclo. A presença de um ciclo C faz com que G não seja um árvore

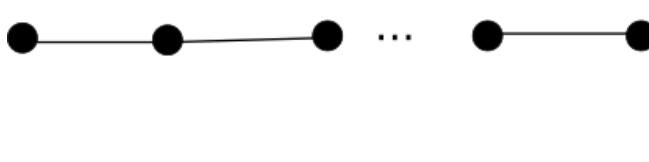
(volta) $q \rightarrow p = !p \rightarrow !q$
 G não é árvore $\rightarrow \exists$ aresta não ponte

Para G não ser uma árvore, deve existir um ciclo em G . Logo, todas as arestas pertencentes ao ciclo não são pontes.

Prop5: A soma dos graus de uma árvore de n vértices não depende da lista de graus, sendo dada por $2n - 2$

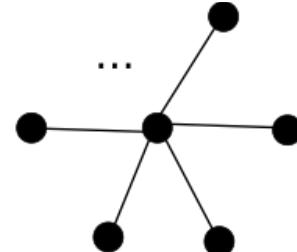
$$\sum_{i=1}^n d(v_i) = 2|E| = 2(n-1) = 2n - 2$$

Ex:



$$L1 = (1, 1, 2, 2, 2, \dots, 2)$$

$n - 2$



$$L2 = (1, 1, \dots, 1, n-1)$$

$n - 1$

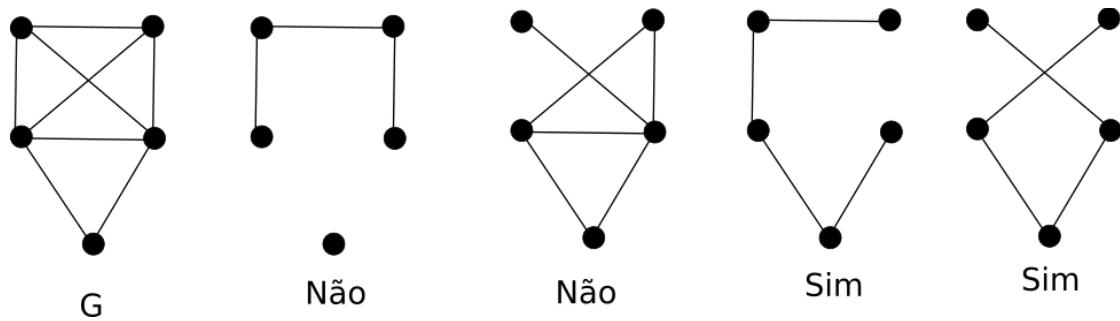
Dentre as árvores que podem ser extraídas a partir de um grafo, há um tipo que é sem dúvida o mais importante: as árvores geradoras. Em resumo, uma árvore geradora representa uma forma resumida de representar um grafo G pois há um e apenas um caminho entre qualquer par de vértices

Prop: Seja $G = (V, E)$ um grafo básico simples e \bar{d} o grau médio dos vértices. Se $\bar{d} \geq 2$, então há pelo menos um ciclo em G

Considere uma árvore T . Então a soma dos graus de T é $2n - 2$. Insira uma aresta qualquer em T , gerando $T + e$. Isso contribui em 2 unidades na soma dos graus fazendo que ela seja igual a $2n$, ou seja, $\bar{d} = 2$. Se forem adicionadas mais que 1 aresta, esse número certamente será maior que 2.

Def: Árvore geradora (spanning tree)

Seja $G = (V, E)$ um grafo. Dizemos que $T = (V, E_T)$ é uma árvore geradora de G se T é um subgrafo de G que é uma árvore (ou seja tem que conectar todos os vértices)



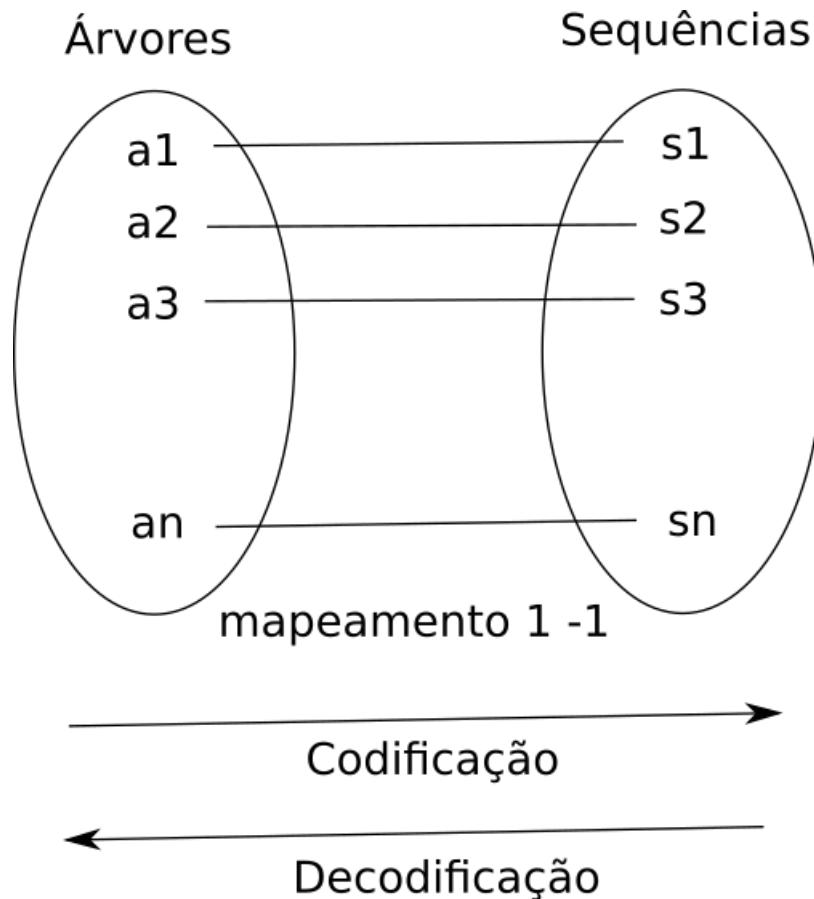
Como pode ser visto, um grafo G admite inúmeras árvores geradoras. A pergunta que surge é: Quantas árvores geradoras existem num grafo $G = (V, E)$ de n vértices?

Para entender a resposta dessa pergunta iremos discutir o código de Prüfer.

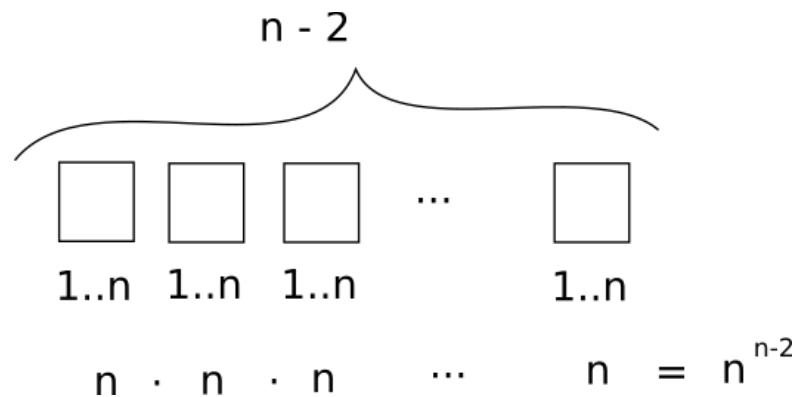
Código de Prüfer

Dada uma rotulação dos vértices, é um código que identifica unicamente cada possível árvore de $n > 2$ vértices. É como se fosse o CPF de uma árvore, cada uma diferente tem o seu código

Foi descoberto que existe uma bijeção entre o conjunto das árvores de n vértices e o conjunto de sequências de inteiros de tamanho $n - 2$



Cada sequencia é composta por $n - 2$ inteiros que podem assumir valores de 1 até n



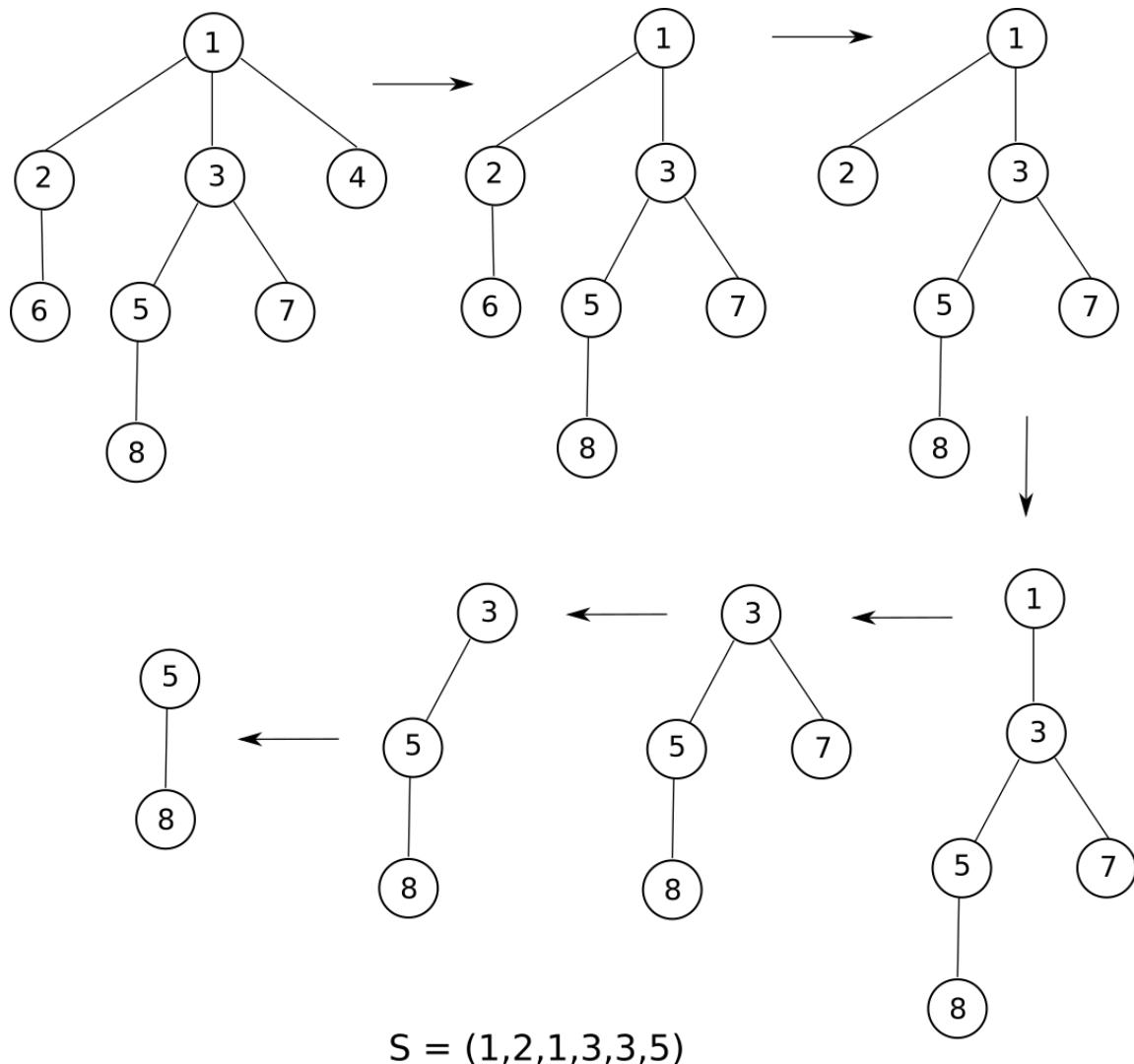
Teorema de Cayley: O grafo K_n tem n^{n-2} árvores geradoras
 Ou seja, isso significa que o número de árvores de n vértices é n^{n-2}

A seguir, iremos apresentar os algoritmos para codificação e decodificação.

Algoritmo: Codificação de Prüfer

Entrada: árvore T

- 1) Rotular cada vértice da árvore com um número inteiro distinto
- 2) O vértice v incidente à folha de menor rótulo é único. Identificar v.
 (v é a resposta para a seguinte pergunta: quem é o pai da folha de menor rótulo?)
- 3) O rótulo de v é adicionado a uma lista S e a folha é removida da árvore
- 4) Repetir passos 2 e 3 até que reste apenas um único vértice ou aresta



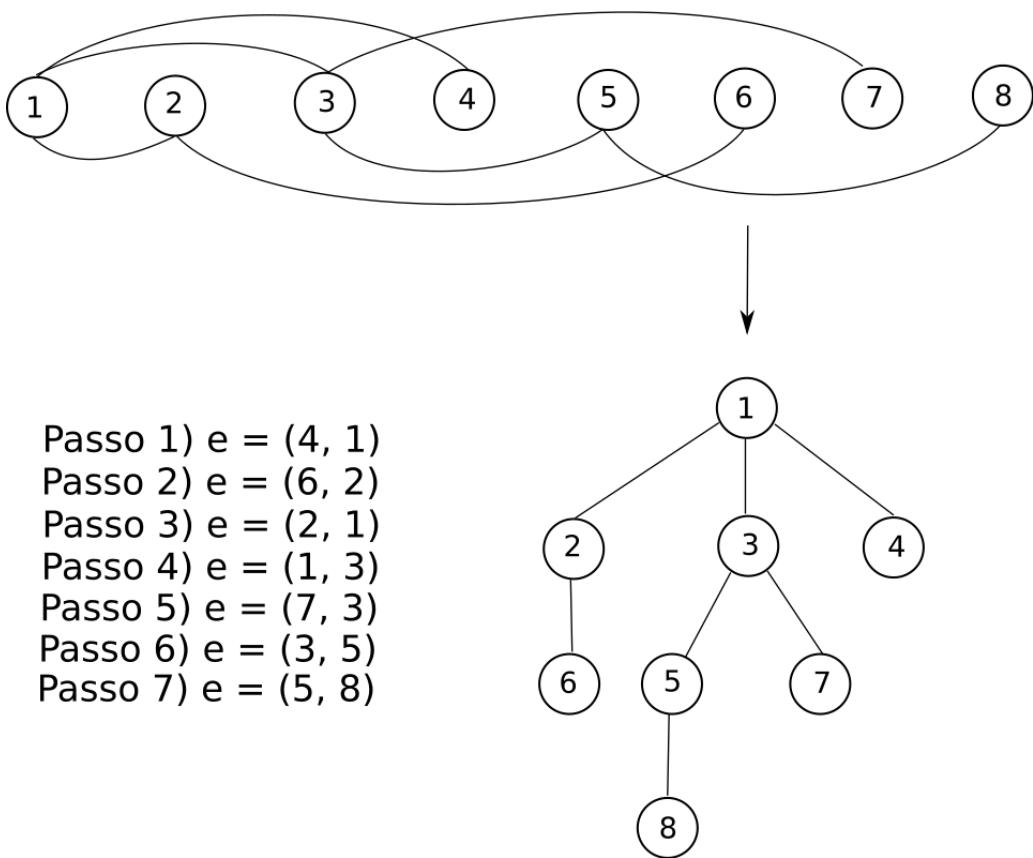
Algoritmo: Decodificação de Prüfer

Entrada: Código C

- 1) Representar a árvore inicial como o grafo nulo de n vértices
- 2) Definir o conjunto $S = \{1, 2, 3, \dots, n\}$
- 3) Buscar em S o menor inteiro que não está no código C . Chamaremos esse número de s_{min}
- 4) Adicionar à árvore T a aresta formada pelo elemento mais a esquerda de C e s_{min}
- 5) Excluir s_{min} de S e o elemento mais a esquerda de C , formando novos S e C
- 6) Repetir os passos 3 a 5 até que não exista mais elementos em C
- 7) Por fim, adicione a T a aresta correspondente aos dois vértices restantes em S

$$C = (1, 2, 1, 3, 3, 5)$$

$$S = \{1, 2, 3, 4, 5, 6, 7, 8\}$$



Árvores Geradoras Mínimas (Minimum Spanning Trees - MST's)

Até o presente momento, estamos lidando com grafos sem pesos nas arestas. Isso significa que dado um grafo G , temos inúmeras formas de obter uma árvore geradora T (vimos que existem muitas dessas árvores num grafo de n vértices). A partir de agora, iremos lidar com grafos ponderados, ou seja, grafos em que as arestas possuem um peso/custo de conexão. O objetivo da seção em questão consiste em fornecer algoritmos para resolver o seguinte problema: dado um grafo ponderado G , obter dentre todas as árvores geradoras possíveis, aquela com o menor peso.

Definição do problema: Dado $G = (V, E, w)$, onde $w: E \rightarrow \mathbb{R}^+$ (peso da aresta e), obter a árvore geradora T que minimiza o seguinte critério:

$$w(T) = \sum_{e \in T} w(e) \quad (\text{soma dos pesos das arestas que compõem a árvore})$$

Obs: A matriz de adjacência de um grafo ponderado é dada por:

$$A_{i,j} = \begin{cases} w_{ij}, & i \leftarrow \rightarrow j \\ \infty, & c.c \end{cases}$$

Exemplo: Interligação banda larga dos bairros de São Carlos (NET)

Ideia geral: a cada passo escolher a aresta de menor peso que seja segura

Aresta segura = aresta que ao ser inserida não faz a árvore deixar de ser árvore

Como determinar se uma aresta é segura?

Cada algoritmo propõe suas especificações próprias para isso

Algoritmo de Kruskal

Objetivo: escolher a cada passo a aresta de menor peso que não forme um ciclo

Entrada: $G = (V, E, w)$ conexo

Saída: $T = (V, E_T)$

1. Defina $T = (V, E_T)$ como o grafo nulo de n vértices

2. Enquanto $|E_T| < n - 1$

a) $T = T + \{e\}$, com e sendo a aresta de menor peso em G que não forma m ciclo em T

b) $E = E - \{e\}$

Segundo Cormen, uma metodologia para detecção de ciclos pode ser implementada utilizando a seguinte ideia: inicialmente cada vértice é colocado num grupo distinto e cada vez que uma aresta é inserida, os dois vértices extremidades passam a fazer parte do mesmo grupo. Assim, arestas que ligam vértices do mesmo grupo, formam ciclos e devem ser evitadas.

Para isso, Cormen apresenta 3 primitivas básicas:

`make_set(v)`: cria um grupo contendo um único vértice, v

`find_set(v)`: retorna o grupo a que o vértice v pertence

`union(u, v)`: faz a união dos grupos de u e v , criando um único grupo

PSEUDOCÓDIGO (Cormen)

`MST_Kruskal(G, w)` // Entrada: grafo G ponderados

{

$A = \emptyset$

 for each $v \in V$

`make_set(v)`

 sort edges into nondecreasing order

 for each $e \in E$ (ordered)

 {

 if (`find_set(u)` ≠ `find_set(v)`)

 {

$A = A \cup \{(u, v)\}$

`union(u, v)`

 }

 }

}

Algoritmo Guloso (segue estratégia de resolver problemas fazendo sempre a escolha ótima em cada passo. Nesse caso, sempre tenta a aresta de menor peso)

A seguir iremos realizar um trace do algoritmo (simulação passo a passo). Para isso iremos

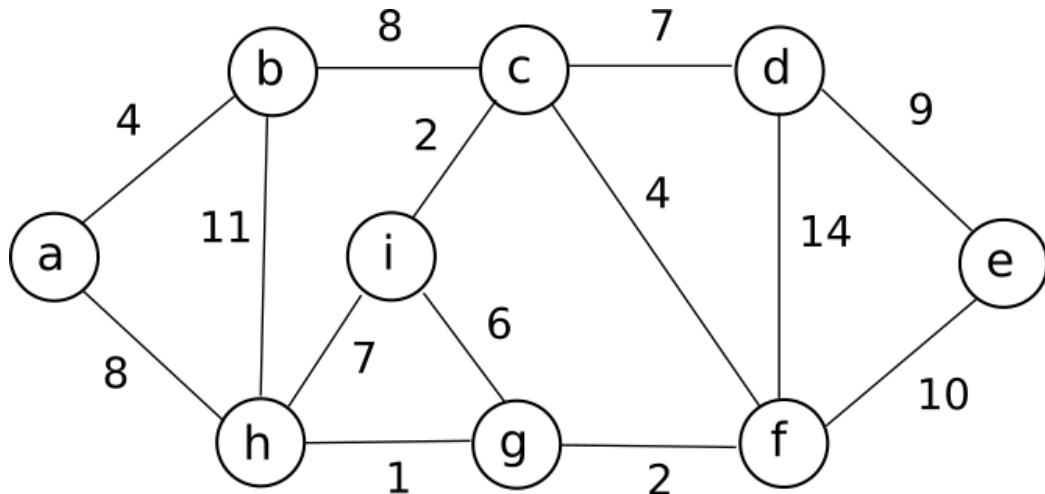
considerar a seguinte notação:

E^- : conjunto das arestas de peso mínimo não seguras ($\text{find_set}(u) = \text{find_set}(v)$)

E^+ : conjunto das arestas de peso mínimo seguras ($\text{find_set}(u) \neq \text{find_set}(v)$)

e_k : aresta escolhida no passo k

Ex: Suponha que os vértices representem bairros e as arestas com pesos os custos de interligação desses bairros (fibra ótica)

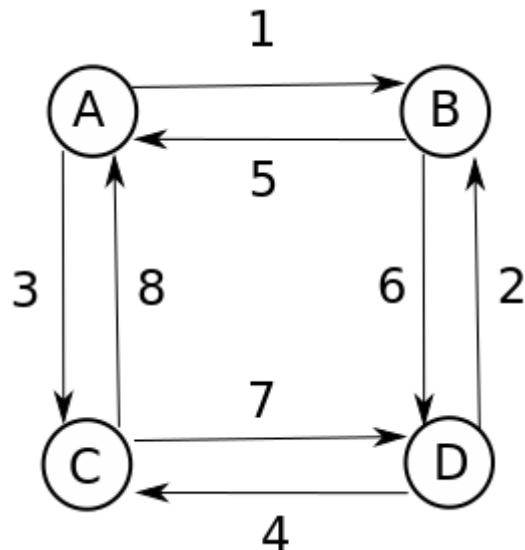


$$E = [1, 2, 2, 4, 4, 6, 7, 7, 8, 8, 9, 10, 11, 14]$$

k	E^-	E^+	e_k
1	-	$\{(g,h)\}$	(g,h)
2	-	$\{(c,i), (f,g)\}$	(c,i)
3	-	$\{(g,f)\}$	(g,f)
4	-	$\{(a,b), (c,f)\}$	(a,b)
5	-	$\{(c,f)\}$	(c,f)
6	$\{(g,i)\}$	-	-
7	$\{(h,i)\}$	$\{(c,d)\}$	(c,d)
8	-	$\{(a,h), (b,c)\}$	(a,h)
9	$\{(b,c)\}$	-	-
10	-	$\{(d,e)\}$	(d,e)

Observações:

- 1) Padrão de crescimento: o subgrafo gerado a cada iteração não é uma árvore (o método segue adicionando arestas a T_k , mas só podemos garantir que temos uma árvore após a inserção da última aresta, ou seja, T_k só é de fato uma árvore na última iteração)
- 2) O algoritmo não deixa explícito como evitar a formação de ciclos
- 3) Podem encontrar MST's em componentes de grafos desconexos
- 4) Grafos direcionados (dígrafos) representam um problema



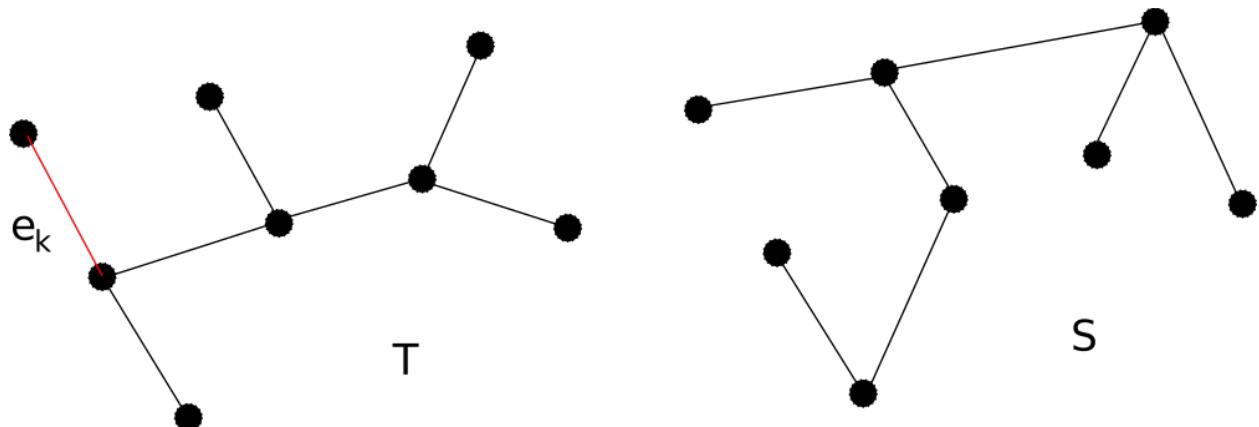
Teorema: Toda árvore T gerada pelo algoritmo de Kruskal é uma MST de G

Garante que o algoritmo sempre funciona e é ótimo

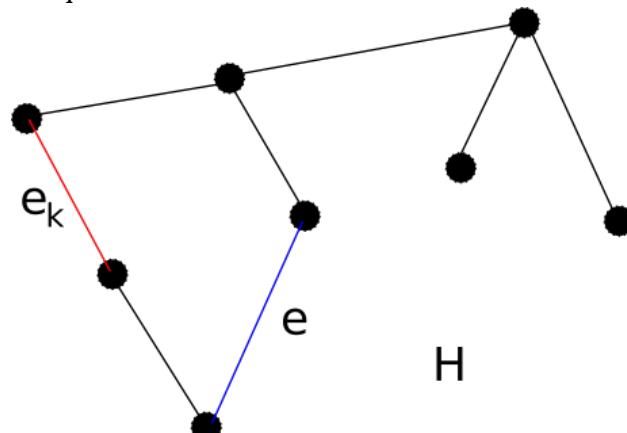
Prova por absurdo

Obs: T é a árvore retornada por Kruskal

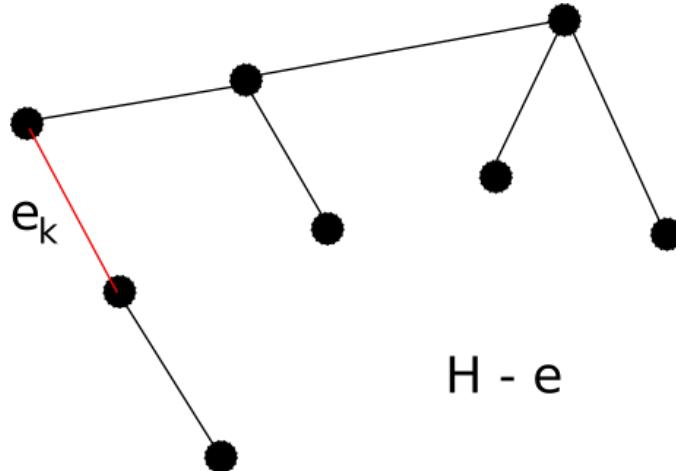
1. Supor que $\exists S \neq T$ tal que $w(S) < w(T)$ (S é uma árvore)
2. Seja $e_k \in T$ a primeira aresta adicionada em T que não está em S (pois árvores são diferentes)



3. Faça $H = (S + e_k)$. Note que H não é mais uma árvore e contém um ciclo



4. Note que no ciclo C , $\exists e \in S$ tal que $e \notin T$ (pois senão C existiria em T). O subgrafo $H - e$ é conexo, possui $n - 1$ arestas e define uma árvore geradora de G .

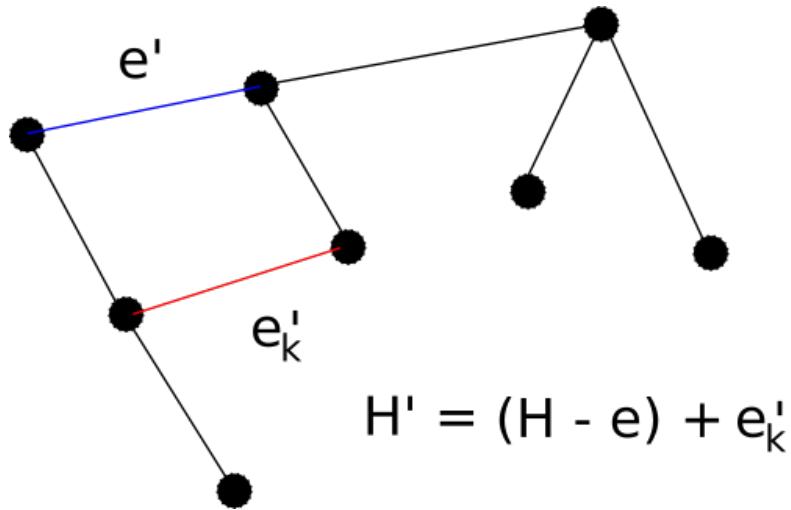


5. Porém, $w(e_k) \leq w(e)$ e assim $w(H - e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, é garantido pela ordenação)

Lista de arestas (após ordenação)



6. Repetindo o processo usado para gerar $H - e$ a partir de S é possível produzir uma sequência de árvores que se aproximam cada vez mais de T .



$$S \rightarrow (H - e) \rightarrow (H' - e') \rightarrow (H'' - e'') \rightarrow \dots \rightarrow T$$

de modo que

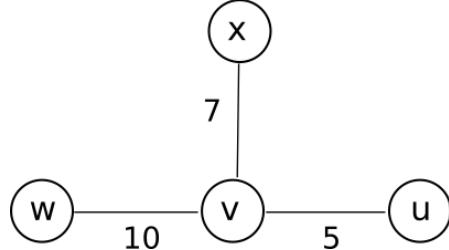
$$w(S) \geq w(H - e) \geq w(H' - e') \geq \dots \geq w(T) \quad (\text{contradição a suposição inicial})$$

Portanto, não existe árvore com peso menor que T , mostrando que T tem peso mínimo. (Não há como S ter peso menor que T)

Algoritmo de Prim

Ideia geral: Inicia em uma raiz r . Enquanto T não contém todos os vértices de G , o algoritmo iterativamente adiciona a T a aresta de menor peso que sai do conjunto dos vértices finalizados (S) e chega no conjunto dos vértices em aberto ($V - S$)

Mas como ele faz isso? Atualizando e mantendo armazenado o menor custo de entrada para qualquer vértice v do grafo



1º entrada de v a ser descoberta é w e seu custo é 10

2º entrada de v a ser descoberta é x e seu custo é 7 que é melhor que 10 (atualiza)

3º entrada de v a ser descoberta é u e seu custo é 5 que é melhor que 7 (atualiza)

Também é considerado um algoritmo guloso (greedy)

Definições de variáveis

$\lambda(v)$: menor custo estimado de entrada para o vértice v (até o presente momento)

$\pi(v)$: predecessor de v na árvore (vértice pelo qual entrei em v)

Q : fila de prioridades dos vértices (maior prioridade = menor $\lambda(v)$)

Como sei que atingi o menor valor de entrada para um vértice v ?

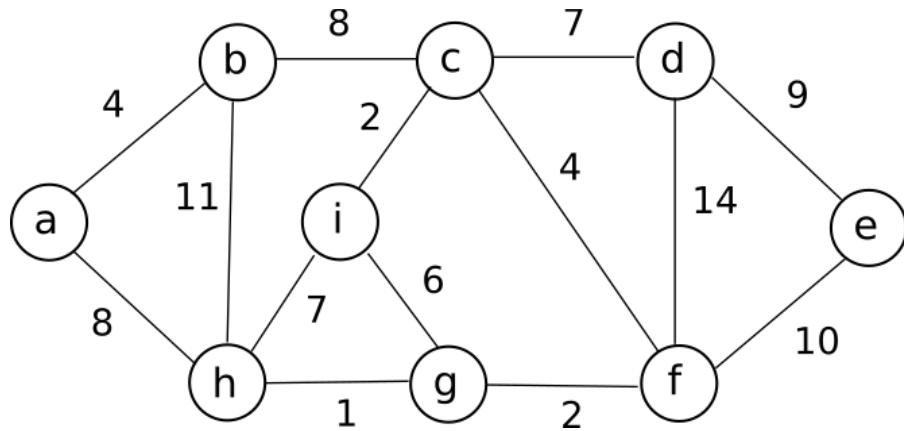
Basta olhar na fila Q . Quando v é removido da fila, seu custo é mínimo ($\lambda(v)$ não muda mais).

Para todo vértice v inserido em S , é garantido que o menor $\lambda(v)$ já foi encontrado.

PSEUDOCÓDIGO

```

MST_Prim(G, w, r) {
    for each  $v \in V$  {
         $\lambda(v) = \infty$ 
         $\pi(v) = \text{nil}$ 
    }
     $\lambda(r) = 0$  (raiz deve iniciar com custo zero pois é primeira a sair da fila)
     $Q = V$  (fila de prioridades inicial é todo conjunto de vértices)
     $S = \emptyset$ 
    while  $Q \neq \emptyset$  {
         $u = \text{ExtractMin}(Q)$  (remove da fila o vértice de menor prioridade)
         $S = S \cup \{u\}$  (já obtive o menor custo de entrada para  $u$ )
        for each  $v \in N(u)$  { (para todo  $v$  vizinho a  $u$ )
            (se não estiver na fila  $Q$ , não pode mais modificar  $\lambda(v)$ )
            if ( $v \in Q$  and  $\lambda(v) > w(u, v)$ ) { (achou porta de entrada de menor custo)
                 $\lambda(v) = w(u, v)$  (atualiza o custo para o menor valor)
                 $\pi(v) = u$  (muda o predecessor, pois achei nova entrada melhor)
            }
        }
    }
}
    
```



OBS: Note que os códigos a seguir fazem exatamente a mesma operação

```

if (  $v \in Q$  and  $\lambda(v) > w(u, v)$  ) {
     $\lambda(v) = w(u, v)$ 
     $\pi(v) = u$ 
}
if  $v \in Q$  {
     $\lambda(v) = \min\{\lambda(v), w(u, v)\}$ 
    if (  $\lambda(v)$  was changed )
         $\pi(v) = u$ 
}

```

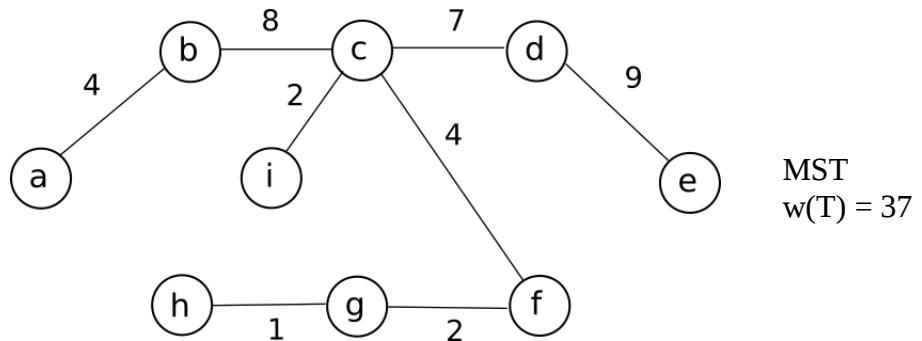
Fila

	a	b	c	d	e	f	g	h	i
$\lambda^{(0)}(v)$	0	∞							
$\lambda^{(1)}(v)$		4	∞	∞	∞	∞	∞	8	∞
$\lambda^{(2)}(v)$			8	∞	∞	∞	∞	8	∞
$\lambda^{(3)}(v)$				7	∞	4	∞	8	2
$\lambda^{(4)}(v)$				7	∞	4	6	7	
$\lambda^{(5)}(v)$				7	10		2	7	
$\lambda^{(6)}(v)$				7	10			1	
$\lambda^{(7)}(v)$					9				

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b) = \min\{\lambda(b), w(a, b)\} = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\lambda(h), w(a, h)\} = \min\{\infty, 8\} = 8$	$\pi(b) = s$ $\pi(h) = s$
b	{c, h}	$\lambda(c) = \min\{\lambda(c), w(b, c)\} = \min\{\infty, 8\} = 8$ $\lambda(h) = \min\{\lambda(h), w(b, h)\} = \min\{8, 11\} = 8$	$\pi(c) = b$ ---
c	{d, f, i}	$\lambda(d) = \min\{\lambda(d), w(c, d)\} = \min\{\infty, 7\} = 7$ $\lambda(f) = \min\{\lambda(f), w(c, f)\} = \min\{\infty, 4\} = 4$ $\lambda(i) = \min\{\lambda(i), w(c, i)\} = \min\{\infty, 2\} = 2$	$\pi(d) = c$ $\pi(f) = c$ $\pi(i) = c$
i	{h, g}	$\lambda(h) = \min\{\lambda(h), w(i, h)\} = \min\{8, 7\} = 7$ $\lambda(g) = \min\{\lambda(h), w(i, g)\} = \min\{\infty, 6\} = 6$	$\pi(h) = i$ $\pi(g) = i$

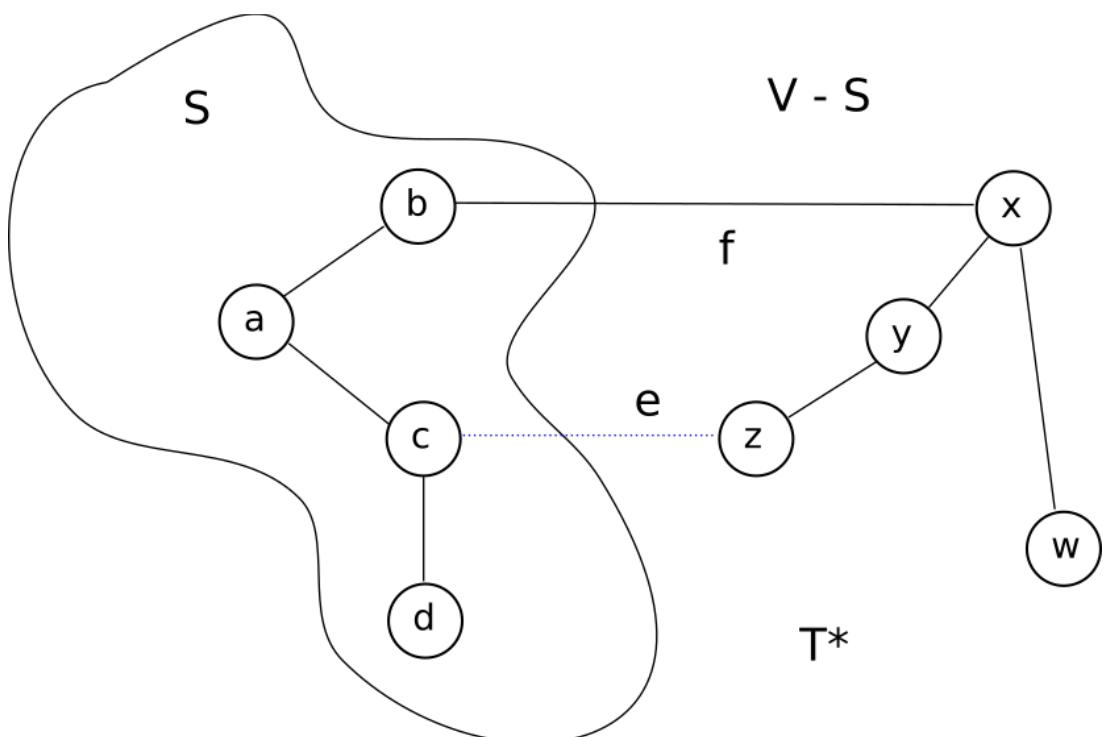
f	{d, e, g}	$\lambda(g) = \min\{\lambda(d), w(f, d)\} = \min\{7, 14\} = 7$	---	$\pi(e) = f$
g	{h}	$\lambda(e) = \min\{\lambda(e), w(f, e)\} = \min\{\infty, 10\} = 10$	$\pi(e) = f$	$\pi(h) = g$
h	\emptyset	$\lambda(h) = \min\{\lambda(h), w(g, h)\} = \min\{7, 1\} = 1$	---	---
d	{e}	$\lambda(e) = \min\{\lambda(e), w(d, e)\} = \min\{10, 9\} = 9$	$\pi(e) = d$	---
e	\emptyset	---	---	---



Mapa de predecessores (árvore final)

v	a	b	c	d	e	f	g	h	i
$\pi(v)$	--	a	b	c	d	c	f	g	c
$\lambda(v)$	0	4	4	8	7	9	4	2	2

Propriedade do corte: Seja $G = (V, E, w)$ um grafo, S um subconjunto qualquer de V e $e \in E$ uma aresta de menor custo com exatamente uma extremidade em S . Então, a MST de G contém e .

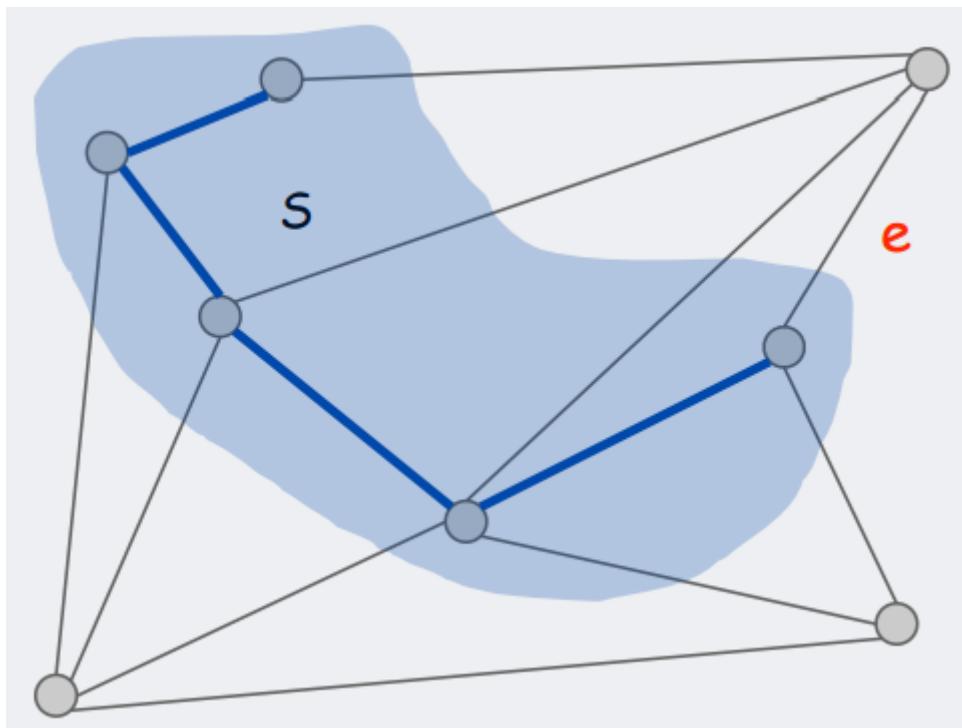


Prova por contradição:

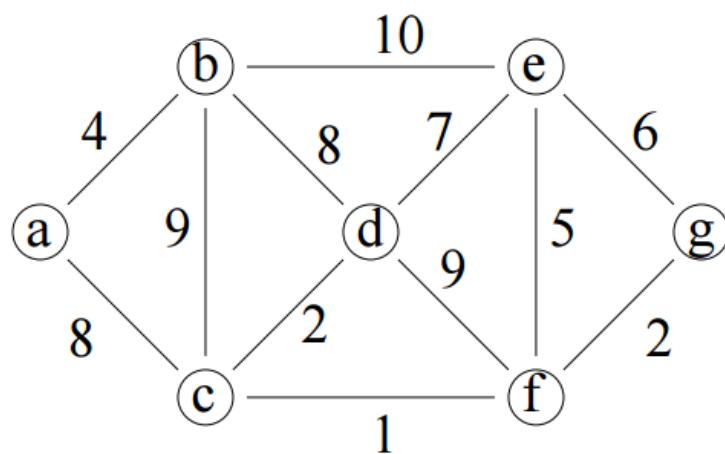
Suponha que $e \notin T^*$. Ao adicionar e em T^* cria-se um único ciclo C . Para que $T^* - e$ fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (senão T^* seria desconexo). Então $T = T^* + e - f$ também é árvore geradora. Como, $w(e) < w(f)$, segue que T tem peso mínimo. Portanto, T^* não é MST de G .

Teorema: A árvore T obtida pelo algoritmo de Prim é uma MST de G .

Seja S o subconjunto de vértices de G na árvore T (definido pelo algoritmo). O algoritmo de Prim adiciona em T a cada passo a aresta de menor custo com apenas um vértice extremidade em S . Portanto, pela propriedade do corte, toda aresta e adicionada pertence a MST de G .



Exercício: encontre a MST do grafo a seguir utilizando o algoritmo de Prim. Mostre o trace completo (passo a passo) do método



Busca em grafos

Importância: como navegar em grafos de maneira determinística de modo a percorrer um conjunto de dados não estruturado (não há linhas, colunas, índices, etc)

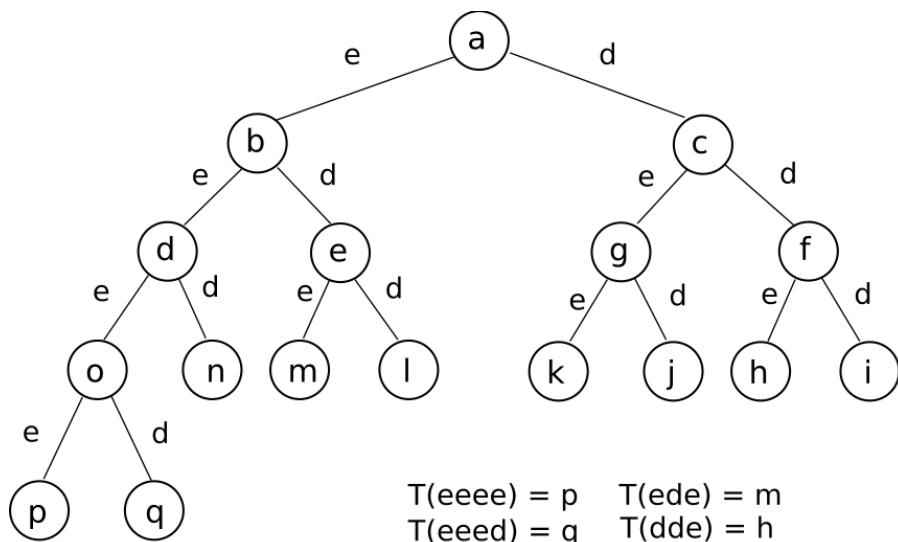
Objetivo: acessar/recuperar todos os elementos do conjunto V

Questões

De quantas maneiras podemos fazer isso? Como? Qual a melhor maneira?

Relação entre busca em grafos e árvores

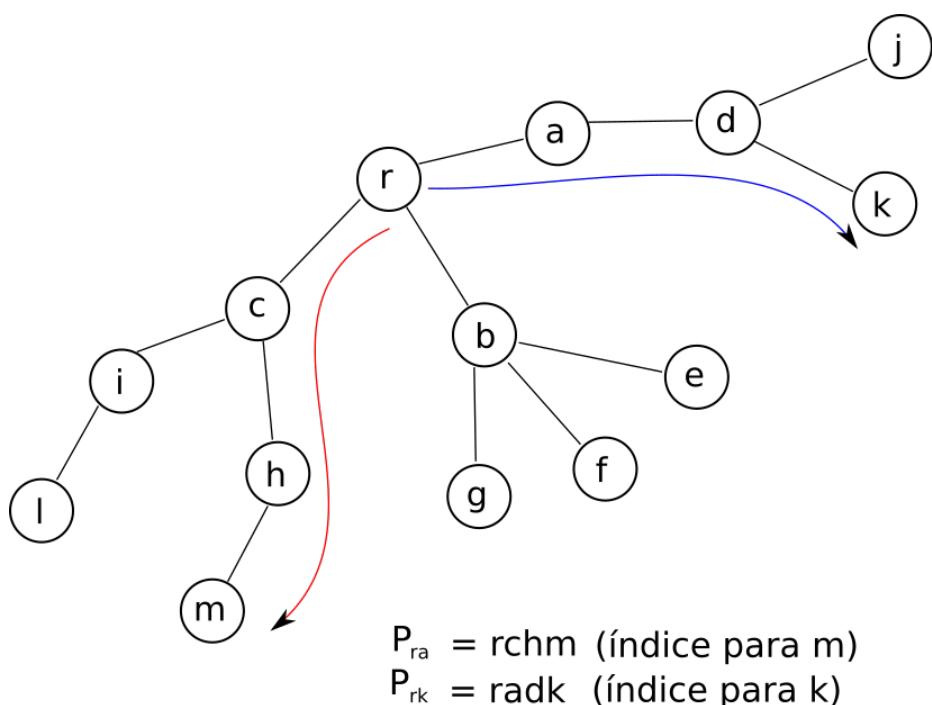
Buscar elementos num grafo G é basicamente o processo de extrair uma árvore T a partir de G. Mas porque? Como busca se relaciona com uma árvore? Isso vem de uma das propriedades das árvores
Numa arvore existe um único caminho entre 2 vértices u, v. Considere uma árvore binária



índices identificam unicamente os elementos

Pode-se criar um esquema de indexamento baseado nos nós a esquerda e a direita. Cada elemento do conjunto possui um índice único que o recupera.

No caso de árvores genéricas, o caminho faz o papel do índice único



Portanto, dado um grafo G, extrair uma árvore T com raiz r a partir dele, significa indexar unicamente cada elemento do conjunto.

Busca em Largura (Breadth-First Search – BFS)

Ideia geral: a cada novo nível descoberto, todos os vértices daquele nível devem ser visitados antes de prosseguir para o próximo nível

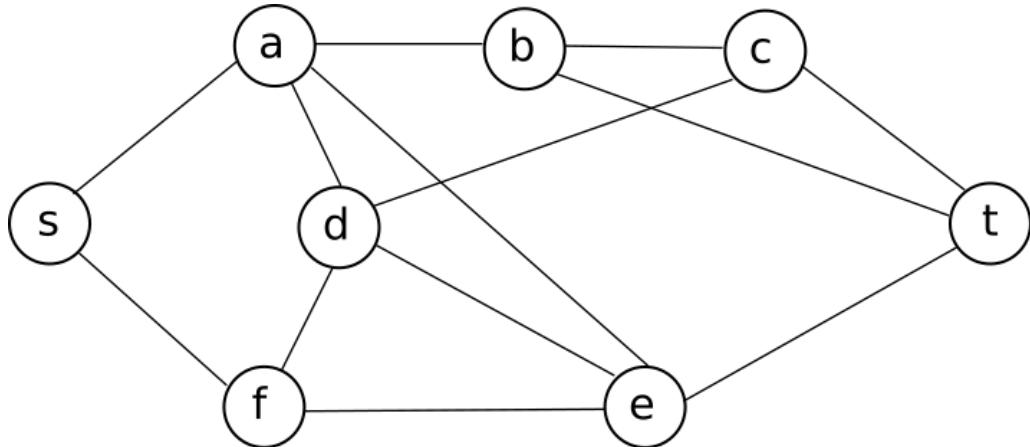
Definição das variáveis (pseudo-código)

- i) v. color : status do vértice v (existem 3 possíveis valores)
 - a) WHITE: vértice v ainda não descoberto (significa que v ainda não entrou na fila Q)
 - b) GRAY: vértice já descoberto (significa que v está na fila Q)
 - c) BLACK: vértice finalizado (significa que v já saiu da fila Q)
- ii) $\lambda(v)$: armazena a menor distância de v até a raiz
- iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)
- iv) Q: Fila (FIFO)
 - 2 primitivas
 - a) pop: remove elemento do início da fila
 - b) push: adiciona um elemento no final da fila

PSEUDOCÓDIGO

```
BFS(G, s)
{
  for each  $v \in V - \{s\}$ 
  {
    v.color = WHITE
     $\lambda(v) = \infty$ 
     $\pi(v) = \text{nil}$ 
  }
  s.color = GRAY
   $\lambda(s) = 0$ 
   $\pi(s) = \text{nil}$ 
  Q =  $\emptyset$ 
  push(Q, s)
  while  $Q \neq \emptyset$ 
  {
    u = pop(Q)
    for each  $v \in N(u)$  // para todo vizinho de u
    {
      if v.color == WHITE // se ainda não passei por aqui, processo vértice v
      {
         $\lambda(v) = \lambda(u) + 1$  // v é descendente de u então distância +1
         $\pi(v) = u$ 
        v.color = GRAY
        push(Q, v) // adiciona v no final da fila
      }
    }
    u. color = BLACK // Após explorar todos vizinhos de u, finalizo u
  }
}
```

O algoritmo BFS recebe um grafo não ponderado G e retorna uma árvore T , conhecida como BFS-tree. Essa árvore possui uma propriedade muito especial: ela armazena os menores caminhos da raiz s a todos os demais vértices de T (menor caminho de s a v , $\forall v \in V$)



Trace do algoritmo BFS

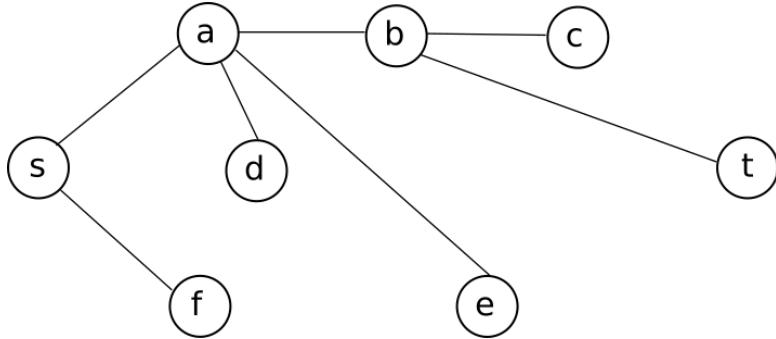
i	$u = \text{pop}(Q)$	$V' = \{ v \in N(u) \mid v.\text{color} = \text{WHITE} \}$	$\lambda(v)$	$\pi(v)$
0	s	{a, f}	$\lambda(a) = \lambda(s) + 1 = 1$ $\lambda(f) = \lambda(s) + 1 = 1$	$\pi(a) = s$ $\pi(f) = s$
1	a	{b, d, e}	$\lambda(b) = \lambda(a) + 1 = 2$ $\lambda(d) = \lambda(a) + 1 = 2$ $\lambda(e) = \lambda(a) + 1 = 2$	$\pi(b) = a$ $\pi(d) = a$ $\pi(e) = a$
2	f	\emptyset	---	---
3	b	{c, t}	$\lambda(c) = \lambda(b) + 1 = 3$ $\lambda(t) = \lambda(b) + 1 = 3$	$\pi(c) = b$ $\pi(t) = b$
4	d	\emptyset	---	---
5	e	\emptyset	---	---
6	c	\emptyset	---	---
7	t	\emptyset	---	---

FILA

$$\begin{aligned}
 Q^{(0)} &= [s] \\
 Q^{(1)} &= [a, f] \\
 Q^{(2)} &= [f, b, d, e] \\
 Q^{(3)} &= [b, d, e] \\
 Q^{(4)} &= [d, e, c, t] \\
 Q^{(5)} &= [e, c, t] \\
 Q^{(6)} &= [c, t] \\
 Q^{(7)} &= [t] \\
 Q^{(8)} &= \emptyset
 \end{aligned}$$

Árvore BFS

v	s	a	b	c	d	e	f	t	
$\pi(v)$	---	s	a	b	a	a	s	b	
$\lambda(v)$	0	1	2	3	2	2	1	3	

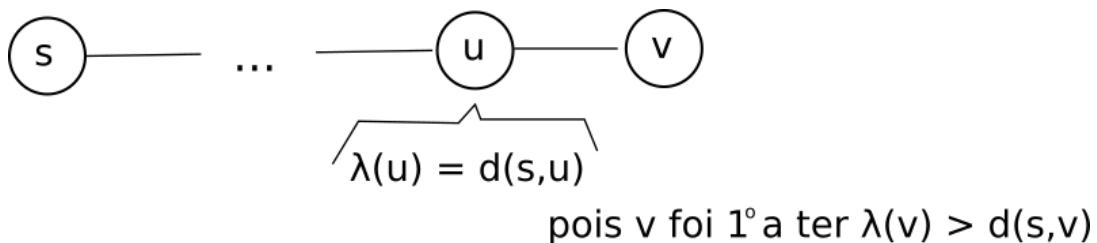


Note que a árvore nada mais é que a união dos caminhos mínimos de s (origem) a qualquer um dos vértices do grafo (destinos). A BFS-tree geralmente não é única, porém todas possuem a mesma profundidade (mínima distância da raiz ao mais distante)

Pergunta: Como podemos implementar um script para computar o diâmetro de um grafo G usando a BFS? Pense em termos de excentricidade

Teorema: A BFS sempre termina com $\lambda(v)=d(s,v)$ para $\forall v \in V$, onde $d(s,v)$ é a distância geodésica (menor distância entre s e v).

1. Sabemos que na BFS $\lambda(v) \geq d(s,v)$
2. Suponha que $\exists v \in V$ tal que $\lambda(v) > d(s,v)$, onde v é o primeiro vértice que isso ocorre ao sair da fila Q
3. Então, existe caminho P_{sv} pois senão $\lambda(v) = d(s,v) = \infty$ (contradiz 2)
4. Se existe P_{sv} então existe um caminho mínimo P_{sv}^*
5. Considere $u \in V$ como predecessor de v em P_{sv}^*



6. Então, $d(s,v) = d(s,u) + 1$ (pois u é predecessor de v)

7. Assim, temos

$$\lambda(v) > d(s,v) = d(s,u) + 1 = \lambda(u) + 1$$

(2) (6) (5)

e portanto $\lambda(v) > \lambda(u) + 1$ (*), o que é uma contradição pois só existem 3 possibilidades quando u sai da fila Q , ou seja, $u = \text{pop}(Q)$

i) v é WHITE: $\lambda(v) = \lambda(u) + 1$ (contradição)

ii) v é BLACK: se isso ocorre significa que v sai da fila Q antes de u , ou seja, $\lambda(v) < \lambda(u)$ (contradição)

iii) v é GRAY: então v foi descoberto por um w removido de Q antes de u , ou seja, $\lambda(w) \leq \lambda(u)$. Além disso, $\lambda(v) = \lambda(w) + 1$. Assim, temos $\lambda(w) + 1 \leq \lambda(u) + 1$, o que finalmente implica em $\lambda(v) \leq \lambda(u) + 1$ (contradição)

Portanto, $\nexists v \in V$ tal que $\lambda(v) > d(s, v)$.

Uma pergunta que o algoritmo anterior ainda não responde é: quantos caminhos mínimos existem de s a t ? Em diversas aplicações é interessante saber o número de rotas ótimas no caso de haver mais de uma. Para isso precisamos lançar mão de um algoritmo de Backtracking

Algoritmo: Backtracking com número de caminhos

Entrada: BFS-tree (saída do BFS)

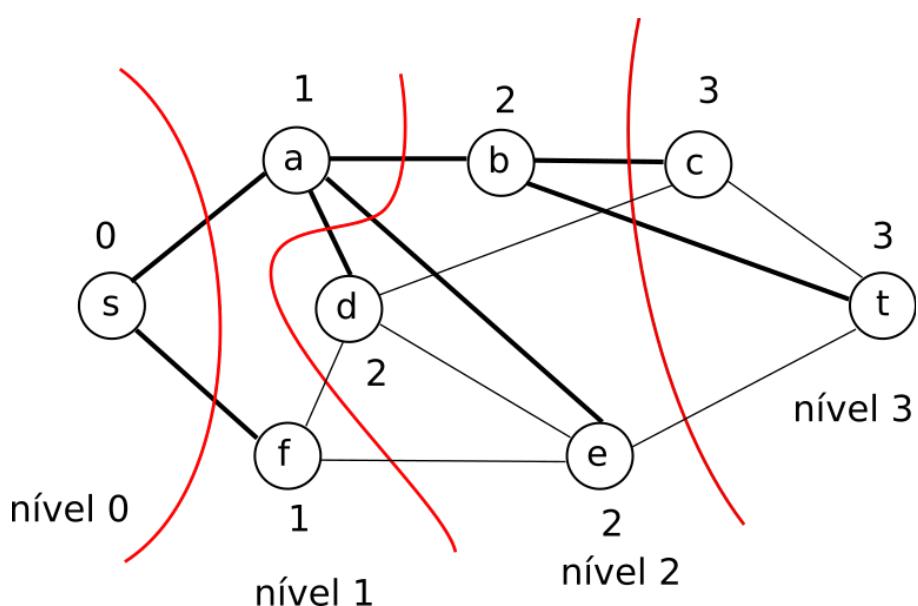
1) Faça $i = \lambda(t)$, $\mu(t) = 1$ e $\forall v \in V (\lambda(v) = \lambda(t) \rightarrow \mu(v) = 0)$ (marcar vértice destino com 1)

2) $\forall v \in V \mid \lambda(v) = i - 1$ faça
 $\mu(v) = \sum \mu(u)$, $\forall u \in N(v) \mid \lambda(u) = i$

3) Se $i = 0$, PARE. Senão, $i = i - 1$ e volte para 2)

Iremos considerar a saída obtida no exemplo anterior. Primeiramente, o algoritmo inicia fazendo:

$i = 3$, $\mu(t) = 1$, $\mu(c) = 0$

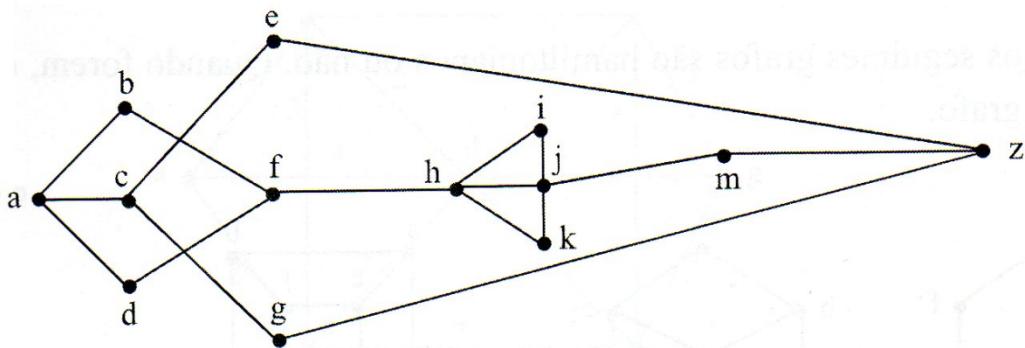


i	$V' = \{v / \lambda(v) = i-1\}$	$\mu(v), v \in V'$
3	{b, d, e}	$\mu(b) = \mu(c) + \mu(t) = 0 + 1 = 1$ $\mu(d) = \mu(c) = 0$ $\mu(e) = \mu(t) = 1$
2	{a, f}	$\mu(a) = \mu(b) + \mu(d) + \mu(e) = 1 + 0 + 1 = 2$ $\mu(f) = \mu(d) + \mu(e) = 0 + 1 = 1$
1	{s}	$\mu(s) = \mu(a) + \mu(f) = 2 + 1 = 3$
0	PARE!	

Portanto, existem 3 caminhos mínimos de s a t: sabt, saet, sfet

Ex: aplicação em jogos
 grafo que representa um mapa
 personagem na posição t
 inimigos na posição s
 cercar personagem por 3 caminhos distintos, level hard

Exercício: Obtenha a BFS-Tree do grafo a seguir. Qual a profundidade da árvore.
 Obtenha o número de caminhos mínimos de a até m.



Busca em Profundidade (Depth-First Search – DFS)

Ideia geral: a cada vértice descoberto, explorar um de seus vizinhos não visitados (sempre que possível). Imita exploração de labirinto, aprofundando sempre que possível.

Definição das variáveis

i) v.d: discovery time (tempo de entrada em v)
 v.f: finishing time (tempo de saída de v)

ii) v. color : status do vértice v (existem 3 possíveis valores)
 a) WHITE: vértice v ainda não descoberto
 b) GRAY: vértice já descoberto
 c) BLACK: vértice finalizado

iii) $\pi(v)$: predecessor de v (onde estava quando descobri v)

iv) Q: Pilha (LIFO)

Para simular a pilha de execução, pode-se utilizar um recurso computacional: Recursão!

Assim, não é necessário implementar de fato essa estrutura de dados (vantagem)
Porém, em casos extremos (tamanho muito grande), recursão pode gerar problemas (desvantagem)

PSEUDOCÓDIGO: (versão recursiva)

```

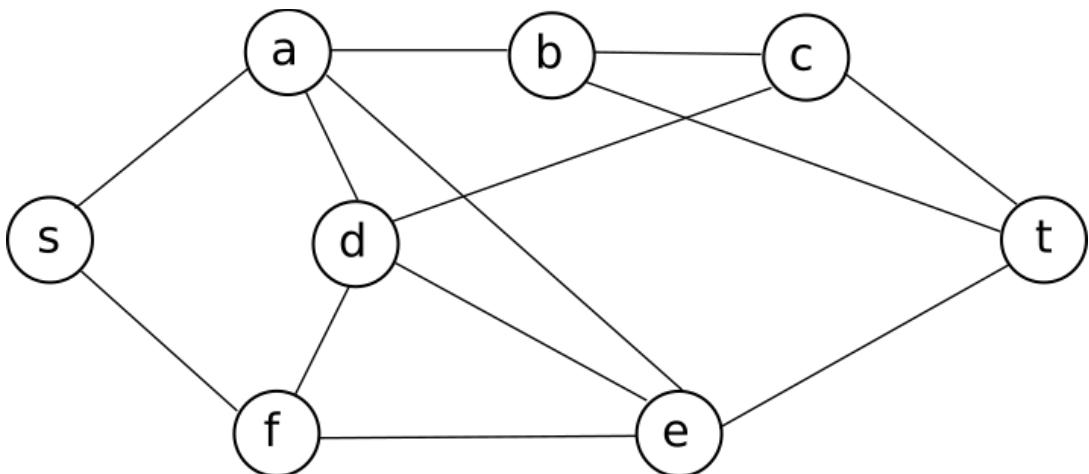
DFS(G, s)
{
    for each  $u \in V$ 
    {
        u.color = WHITE
         $\pi(u) = nil$ 
    }
    time = 0      // variável global para armazenar o tempo
    DFS_visit(G, s)
}

// Função recursiva que é chamada sempre que um vértice é descoberto
DFS_visit(G, u)
{
    time++
    u.d = time
    u.color = GRAY
    for each  $v \in N(u)$ 
    {
        if v.color == WHITE
        {
             $\pi(v) = u$ 
            DFS_visit(G, v)      // chamada recursiva
        }
    }
    u.color = BLACK
    time++
    u.f = time
}

```

Da mesma forma que o algoritmo BFS, esse método recebe um grafo G não ponderado e retorna uma árvore, a DFS_tree.

Ex: Trace do algoritmos



u	u.color	u.d	$V' = \{ v \in N(u) / v.\text{color} = \text{WHITE}\}$	$\pi(u)$	u.f
s	G	1	{a, f}	--	16
a	G	2	{b, d, e}	s	15
b	G	3	{c, t}	a	14
c	G	4	{d, t}	b	13
d	G	5	{e, f}	c	12
e	G	6	{f, t}	d	11
f	G	7	\emptyset	e	8
t	G	9	\emptyset	e	10

BFS

- aspecto espacial
- caminhos mínimos
- Fila

x

DFS

- aspecto temporal
- vértices de corte, ordenação topológica
- Pilha

Propriedades da DFS_tree

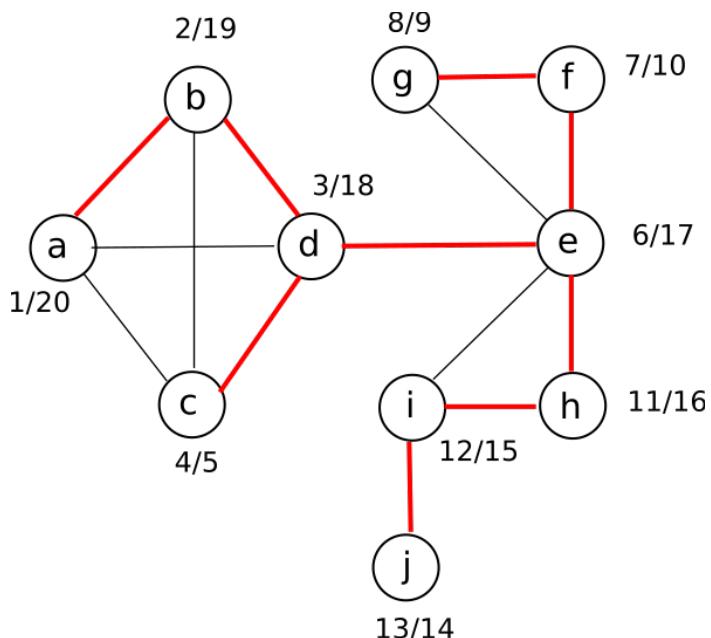
1) A rotulação tem o seguinte significado:

- a) $[u.d, u.f] \subset [v.d, v.f]$: u é descendente de v
- b) $[v.d, v.f] \subset [u.d, u.f]$: v é descendente de u
- c) $[v.d, v.f]$ e $[u.d, u.f]$ são disjuntos: estão em ramos distintos da árvore

2) Após a DFS, podemos classificar as arestas de G como:

- a) tree_edges: $e \in T$
- b) fb_edges (forward-backward edge): $e \notin T$

Def: v é um vértice de corte \Leftrightarrow v tem um filho s tal que \nexists fb_edge ligando s ou qualquer descendente de s a um ancestral de v



Perguntas:

- a) b é vértice de corte? Não pois fb_edge (a, d) liga um sucessor a um antecessor
- b) d é vértice de corte? Sim, pois não há fb_edge entre sucessor e antecessor
- c) e é vértice de corte? Sim, pois não há fb_edge

Ordenação Topológica (TopSort)

É uma aplicação muito importante em computação. Exemplos de aplicações da TopSort incluem a resolução de dependências entre bibliotecas de software, bem como gerenciamento de projetos.

Classe particular de grafos: DAG's (directed acyclic graphs)

Def: A ordenação topológica de um DAG G é uma ordenação linear de modo que se $(u, v) \in E$ então u aparece antes de v

Algoritmo: TopSort(G)

1. Chamar DFS(G) (Obs.: a função DFS deve ser ligeiramente modificada *)
2. Conforme cada vértice é finalizado (BLACK), insira-o no início de uma lista L
3. Retorne a lista L

Obs: Substituir o comando `DFS_visit(G, s)` pelo seguinte trecho de código

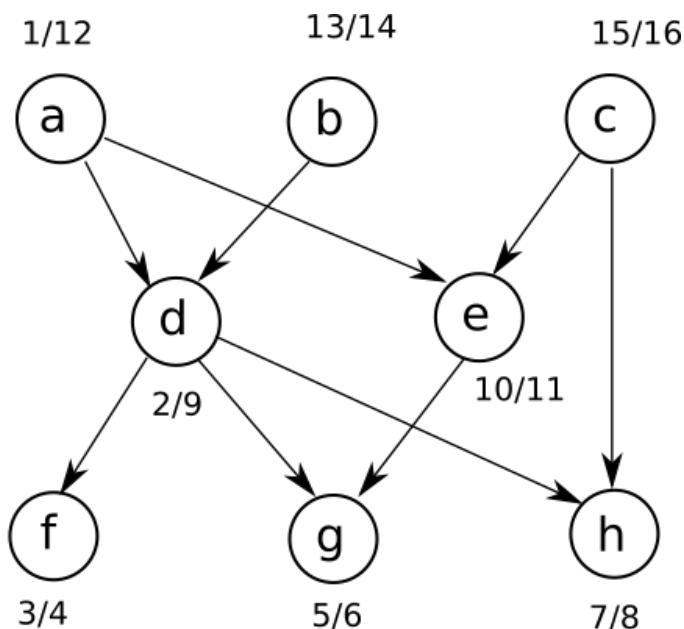
```
* for each  $v \in V$  {  
    if  $v.\text{color} == \text{WHITE}$   
        DFS_visit(G, v)  
}
```

Ex: Considere o seguinte DAG

vértices = tarefas/jobs/processos

aresta $(x, y) = x$ deve estar pronto antes de y iniciar

Qual é uma ordem válida de execução das tarefas? $\text{TopSort}(G)$



$$L = [c, b, a, e, d, h, g, f]$$

Caminhos mínimos em grafos ponderados

Existem basicamente 3 subproblemas principais:

- i) 1 para 1 (Dijkstra) – caminho s-t
- ii) 1 para N (Dijkstra) – árvore de caminhos mínimos
- iii) M para N (Floyd-Warshall) – matriz de distâncias ponto a ponto

Def: Caminho ótimo

Seja $G = (V, E)$ e $w: E \rightarrow R^+$ uma função de custo para as arestas. Um caminho P^* de v_0 a v_n é ótimo se seu peso

$$w(P^*) = \sum_{i=0}^{n-1} w(v_i, v_{i+1}) = w(v_0, v_1) + w(v_1, v_2) + \dots + w(v_{n-1}, v_n) \quad (\text{soma dos pesos das arestas})$$

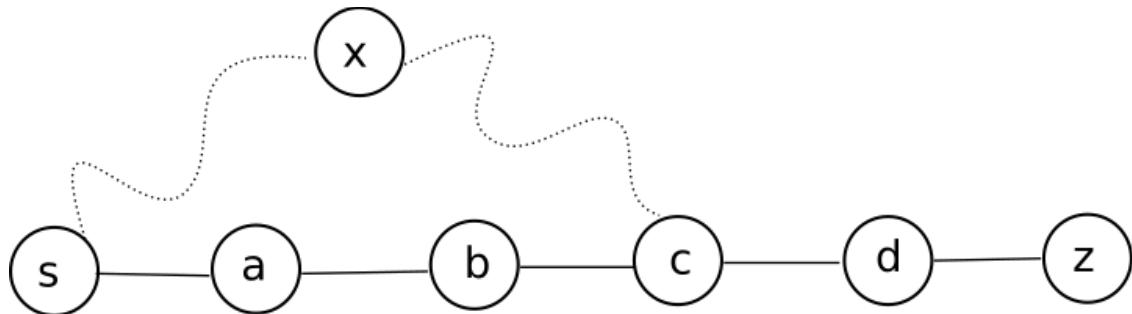
é o menor possível.

Prop1: Todo subcaminho de um caminho ótimo é ótimo

Suponha no grafo a seguir que o caminho ótimo de s a z é $P^* = sabcdz$

Suponha ainda que existe um caminho mínimo de s a c que não passa por a e b mas sim por x, ou seja, $P' = sxcc$

Ora, se isso é verdade, então claramente P^* não é ótimo pois é possível minimizá-lo ainda mais, criando $P = sxcdz$. Assim, subcaminhos de caminhos ótimos são ótimos



Importante resultado na obtenção de algoritmos de programação dinâmica (problema com subestrutura ótima, ou seja, é possível utilizar partes de soluções já obtidas na construção da solução final). Em outras palavras, é isso que permite a utilização de programação dinâmica em algoritmos de caminhos mínimos

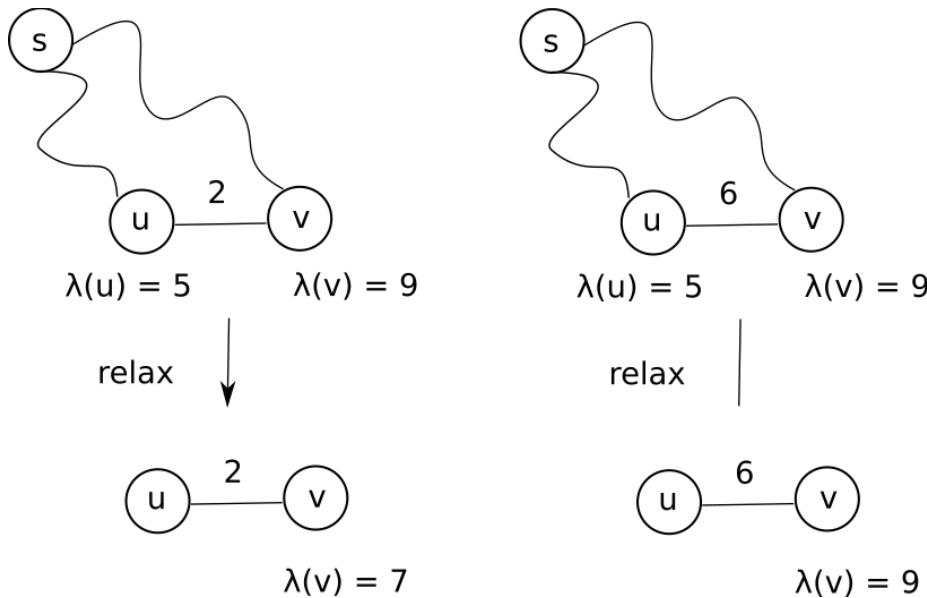
Antes de introduzirmos os algoritmos, iremos apresentar uma primitiva comum a todos eles. Trata-se da função relax, que aplica a operação conhecida como relaxamento a uma aresta de um grafo ponderado.

Primitiva relax

$\text{relax}(u, v, w)$: relaxar a aresta (u, v) de peso w sabendo os valores de $\lambda(u)$ e $\lambda(v)$

Quem é $\lambda(u)$? É o custo atual de sair da origem s e chegar até u
Quem é $\lambda(v)$? É o custo atual de sair da origem s e chegar até v

Ideia geral: é uma boa passar por u para chegar em v sabendo que o custo de ir de u até v é w ?



Obs: A operação $\text{relax}(u, v, w)$ nunca aumenta o valor de $\lambda(v)$, apenas diminui

PSEUDOCODIGO

```

relax(u, v, w)
{
    if  $\lambda(v) > \lambda(u) + w(u, v)$ 
    {
         $\lambda(v) = \lambda(u) + w(u, v)$ 
         $\pi(v) = u$ 
    }
}
relax(u, v, w)
{
     $\lambda(v) = \min\{\lambda(v), \lambda(u) + w(u, v)\}$ 
    if  $\lambda(v)$  was changed
         $\pi(v) = u$ 
}

```

O que varia nos diversos algoritmos para encontrar caminhos mínimos são os seguintes aspectos:

- i) Quantas e quais arestas devemos relaxar?
- ii) Quantas vezes devemos relaxar as arestas?
- iii) Em que ordem devemos relaxar as arestas?

Algoritmo Bellman-Ford

Ideia geral: a cada passo relaxar $\forall e \in E$ em ordem arbitrária, repetindo o processo $|V| - 1$ vezes

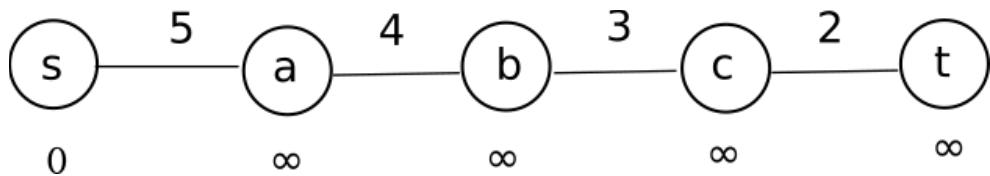
PSEUDOCODIGO

```

Bellman_Ford(G, w, s)
{
    for each  $v \in V$ 
    {
         $\lambda(v) = \infty$ 
         $\pi(v) = \text{nil}$ 
    }
     $\lambda(s) = 0$ 
    for  $i = 1$  to  $|V| - 1$ 
    {
        for each  $e = (u, v)$  in  $E$ 
             $\text{relax}(u, v, w)$ 
    }
}

```

Ex:



Dependendo da ordem que começo a relaxar arestas, pode demorar mais ou menos iterações para o algoritmo convergir. No pior caso, são necessários $|V| |E|$ operações e portanto o algoritmo em questão é considerado bastante ineficiente: $O(|V| |E|)$

A seguir veremos um algoritmo muito mais eficiente para resolver o problema: o algoritmo de Dijkstra. Basicamente, esse algoritmo faz uso de uma política de gerenciamento de vértices baseada em aspectos de programação dinâmica. O que o método faz é basicamente criar uma fila de prioridades para organizar os vértices de modo que quanto menor o custo $\lambda(v)$ maior a prioridade do vértice em questão. Assim, a ideia é expandir primeiramente os menores ramos da árvore de caminhos mínimos, na expectativa de que os caminhos mínimos mais longos usarão como base os subcaminhos obtidos anteriormente. Trata-se de um mecanismo de reaproveitar soluções de subproblemas para a solução do problema como um todo.

Definição das variáveis

- $\lambda(v)$: menor custo até o momento para o caminho $s-v$
- $\pi(v)$: predecessor de v na árvore de caminhos mínimos
- Q: fila de prioridades dos vértices (maior prioridade = menor $\lambda(v)$)

PSEUDOCODIGO

```

Dijkstra(G, w, s)
{
     $\lambda(s)=0$ 
     $\pi(s)=0$ 
    for each  $v \in V$ 
    {
         $\lambda(v)=\infty$ 
         $\pi(v)=nil$ 
    }
    Q = V (fila de prioridades)
    while  $Q \neq \emptyset$ 
    {
        u = ExtractMin(Q)
         $S=S \cup \{u\}$ 
        for each  $v \in N(u)$ 
            relax(u, v, w)
    }
}
    
```

Algoritmos e estruturas de dados

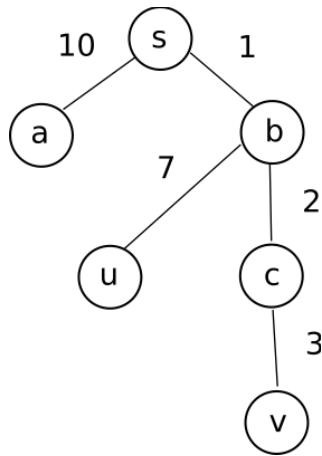
BFS – Fila

DFS – Pilha

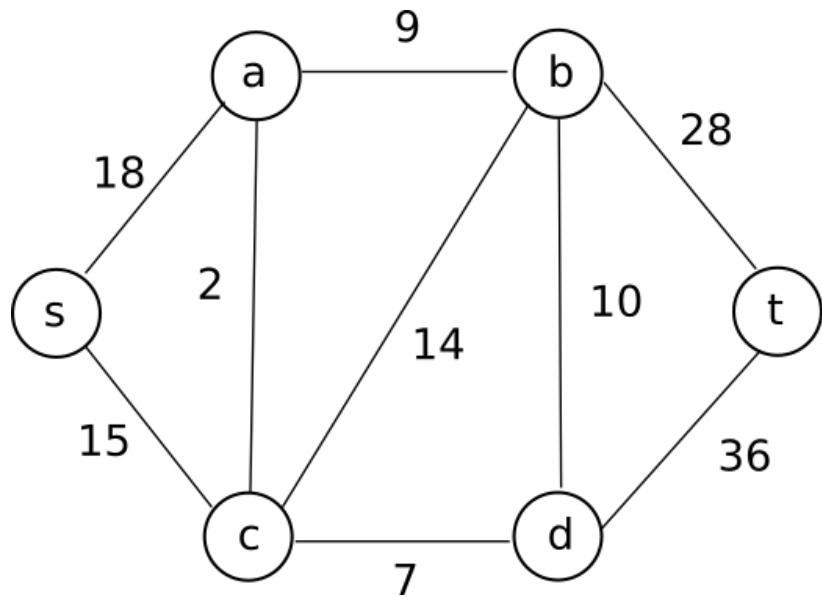
Dijkstra – Fila de prioridades

Obs: Note que o algoritmo de Dijkstra é uma generalização da BFS

Ambos crescem primeiro os menores ramos da árvore. A BFS toda aresta tem mesmo tamanho, no Dijkstra esse tamanho é variável.



Ex:



Fila

	s	a	b	c	d	t	
$\lambda^{(0)}(v)$	0	∞	∞	∞	∞	∞	
$\lambda^{(1)}(v)$		18	∞	15	∞	∞	
$\lambda^{(2)}(v)$			29		22	∞	
$\lambda^{(3)}(v)$				26		∞	
$\lambda^{(4)}(v)$					26	58	
$\lambda^{(5)}(v)$						54	

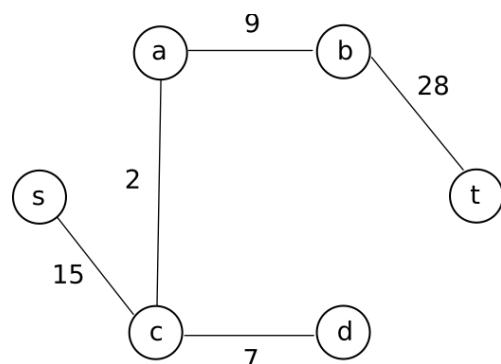
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
s	{a, c}	$\lambda(a) = \min\{\lambda(a), \lambda(s) + w(s, a)\} = \min\{\infty, 18\} = 18$	$\pi(a) = s$
c	{a, b, d}	$\lambda(c) = \min\{\lambda(c), \lambda(s) + w(s, c)\} = \min\{\infty, 15\} = 15$	$\pi(c) = s$
a	{b}	$\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c, a)\} = \min\{18, 17\} = 17$	$\pi(a) = c$
d	{b, t}	$\lambda(b) = \min\{\lambda(b), \lambda(c) + w(c, b)\} = \min\{\infty, 29\} = 29$	$\pi(b) = c$
b	{t}	$\lambda(d) = \min\{\lambda(d), \lambda(c) + w(c, d)\} = \min\{\infty, 22\} = 22$	$\pi(d) = c$
t	\emptyset	$\lambda(b) = \min\{\lambda(b), \lambda(a) + w(a, b)\} = \min\{29, 26\} = 26$	$\pi(b) = a$
		$\lambda(b) = \min\{\lambda(b), \lambda(d) + w(d, b)\} = \min\{26, 32\} = 26$	---
		$\lambda(t) = \min\{\lambda(t), \lambda(d) + w(d, t)\} = \min\{\infty, 58\} = 58$	$\pi(t) = d$
		$\lambda(t) = \min\{\lambda(t), \lambda(b) + w(b, t)\} = \min\{58, 54\} = 54$	$\pi(t) = b$
		---	---

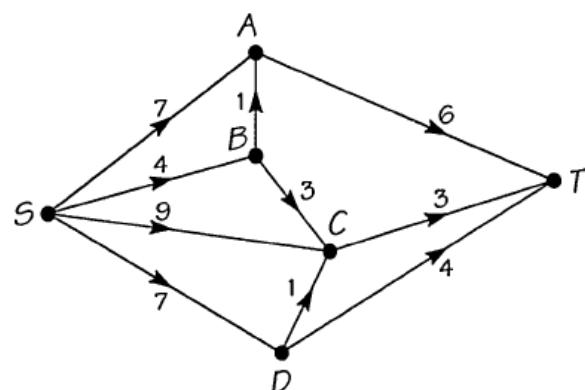
Mapa de predecessores (árvore final)

v		s		a		b		c		d		t	
$\pi(v)$		---		c		a		s		c		b	

Árvore de caminhos mínimos (armazena os menores caminhos de s a todos os demais vértices)



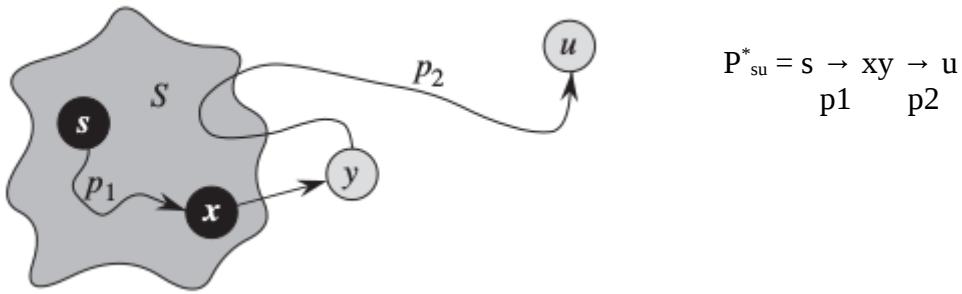
Ex: Utilizando o algoritmo de Dijkstra, construa a árvore de caminhos mínimos (trace completo)



Teorema: O algoritmo de Dijkstra termina com $\lambda(v) = d(s, v), \forall v \in V$

Obs: Note que sempre $\lambda(v) \geq d(s, v)$ (*)

1. Suponha que u seja o 1º vértice para o qual $\lambda(u) \neq d(s, u)$ quando u entra em S .
2. Então, $u \neq s$ pois senão $\lambda(s) = d(s, s) = 0$
3. Assim, existe um caminho P_{su} pois senão $\lambda(u) = d(s, u) = \infty$. Portanto, existe um caminho mínimo P_{su}^*
4. Antes de adicionar u a S , P_{su}^* possui $s \in S$ e $u \in V - S$
5. Seja y o 1º vértice em P_{su}^* tal que $y \in V - S$ e seja x seu predecessor ($x \in S$)



Obs: Note que tanto p_1 quanto p_2 não precisam ter arestas

6. Como $x \in S$, $\lambda(x) = d(s, x)$ e no momento em que ele foi inserido a S , a aresta (x, y) foi relaxada, ou seja:

$$\lambda(y) = \lambda(x) + w(x, y) = d(s, x) + w(x, y) = d(s, y)$$

7. Mas y antecede a u no caminho e como $w: E \rightarrow R^+$ (pesos positivos), temos:

$$d(s, y) \leq d(s, u)$$

e portanto

$$\lambda(y) = d(s, y) \leq d(s, u) \leq \lambda(u)$$

(6) (7) (*)

8. Mas como ambos y e u pertencem a $V - S$, quando u é escolhido para entrar em S temos $\lambda(u) \leq \lambda(y)$

9. Como $\lambda(y) \leq \lambda(u)$ e $\lambda(u) \leq \lambda(y)$ então temos que $\lambda(u) = \lambda(y)$, o que implica em:

$$\lambda(y) = d(s, y) = d(s, u) = \lambda(u)$$

o que gera uma contradição. Portanto $\exists u \in V$ tal que $\lambda(u) \neq d(s, u)$ quando u entra em S .

Dijkstra multisource

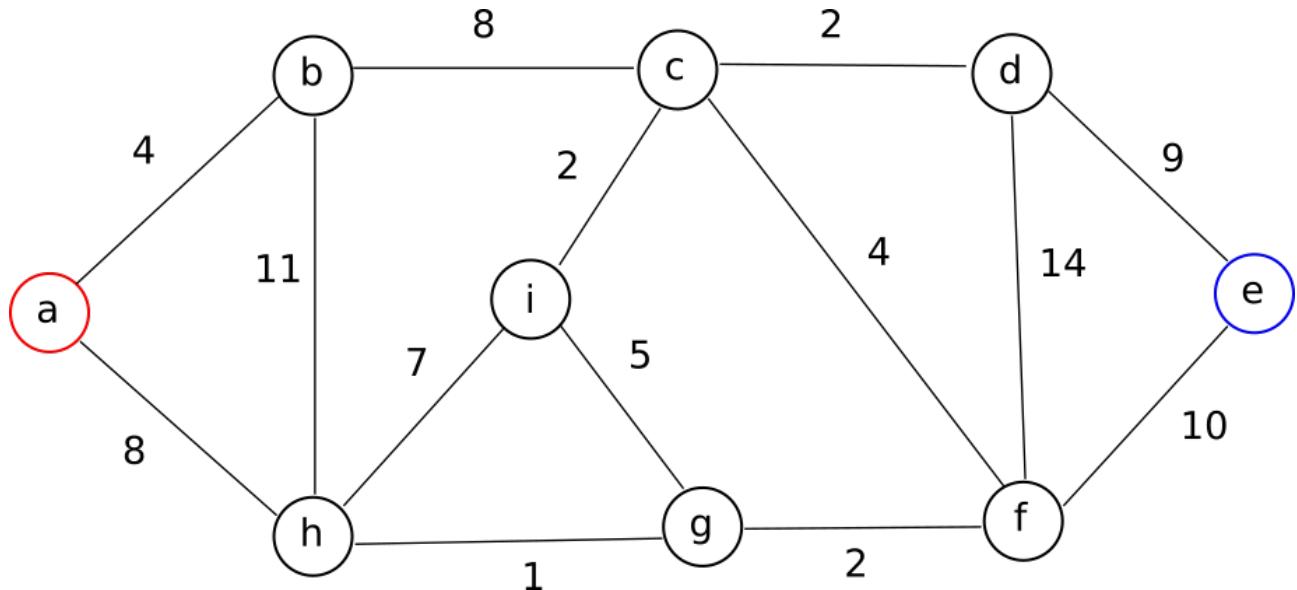
Ideia: utilizar múltiplas sementes/raízes

Processo de competição: cada vértice pode ser conquistado por apenas uma das sementes (pois ao fim, um vértice só pode estar pendurado em uma única árvore)

Durante a execução do algoritmo, nesse processo de conquista, uma semente pode “roubar” um nó de seus concorrentes, oferecendo a ele um caminho menor que o atual

Ao final temos o que se chama de floresta de caminhos ótimos, composta por várias árvores (uma para cada semente)

Cada árvore representa um agrupamento/comunidade.



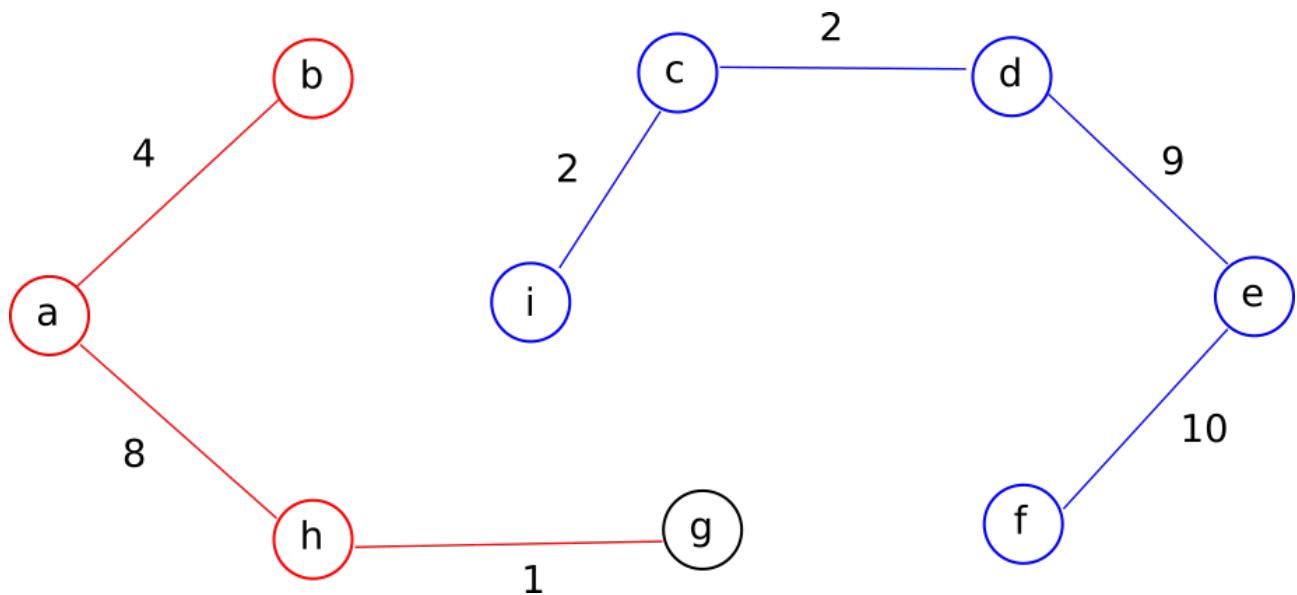
Desejamos encontrar 2 agrupamentos. Para isso, utilizaremos 2 sementes: os vértices A e E. Na prática, isso significa inicializar o algoritmo de Dijkstra com $\lambda(a)=\lambda(e)=0$

Fila

Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, h}	$\lambda(b) = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\infty, 8\} = 8$	$\pi(b) = a$ $\pi(h) = a$
e	{d, f}	$\lambda(d) = \min\{\infty, 9\} = 9$ $\lambda(f) = \min\{\infty, 10\} = 10$	$\pi(d) = e$ $\pi(f) = e$
b	{c, h}	$\lambda(c) = \min\{\infty, 4+8\} = 12$ $\lambda(h) = \min\{8, 4+11\} = 8$	$\pi(c) = b$ -----
h	{i, g}	$\lambda(i) = \min\{\infty, 8+7\} = 15$ $\lambda(g) = \min\{\infty, 8+1\} = 9$	$\pi(i) = h$ $\pi(g) = h$
d	{c, f}	$\lambda(c) = \min\{12, 9+2\} = 11$ $\lambda(f) = \min\{10, 9+14\} = 10$	$\pi(c) = d$ -----
g	{f, i}	$\lambda(f) = \min\{10, 9+14\} = 10$ $\lambda(i) = \min\{15, 9+5\} = 14$	-----
f	{c}	$\lambda(c) = \min\{11, 10+4\} = 11$	-----
c	{i}	$\lambda(i) = \min\{14, 11+2\} = 13$	$\pi(i) = c$
i	\emptyset	-----	-----

Floresta de caminhos ótimos



A heurística A*

É uma técnica aplicada para acelerar a busca por caminhos mínimos em certos tipos de grafos. Pode ser considerado uma generalização do algoritmo de Dijkstra

Ideia: modificar a função que define a prioridade dos vértices

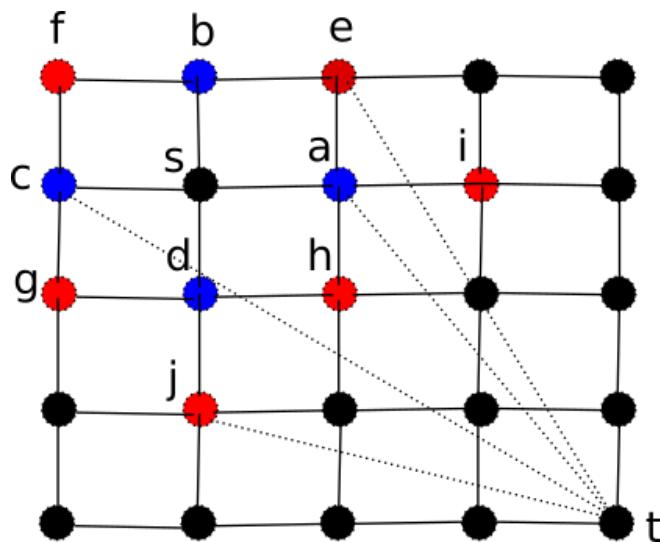
$$\alpha(v) = \lambda(v) + \gamma(v) \quad \text{onde}$$

$\lambda(v)$: custo atual de ir da origem s até v

$\gamma(v)$: custo estimado de v até o destino t (alvo)

Problema: como calcular $\gamma(v)$? Se eu soubesse já teria o caminho mínimo...

- É viável apenas em casos específicos (reticulados, grades 2D e 3D)
- Muito utilizado na IA de jogos



Considere o grafo acima. O vértice s é a origem e o vértice t é o destino. Neste caso temos:

$$\lambda(a) = \lambda(b) = \lambda(c) = \lambda(d) = 1$$

o que significa que no Dijkstra, todos eles teriam a mesma prioridade. Note porém que, utilizando a distância Euclidiana para obter uma estimativa de distância até a origem, temos:

$$\begin{aligned} \gamma(a) &= \gamma(d) = \sqrt{4+9} = \sqrt{13} \\ \gamma(b) &= \gamma(c) = \sqrt{9+16} = \sqrt{25} = 5 \end{aligned}$$

Ou seja, no A*, devemos priorizar a e d em detrimento de b e c uma vez que

$$1 + \sqrt{13} < 1 + 5$$

e portanto a e d saem da fila de prioridades antes. Isso ocorre pois no A* eles são considerados mais importantes. O mesmo ocorre nos demais níveis

$$\lambda(e) = \lambda(f) = \lambda(g) = \lambda(h) = \lambda(i) = \lambda(j) = 2$$

$$\gamma(e) = \sqrt{18} \quad \gamma(j) = \sqrt{10} \quad \gamma(h) = \sqrt{8}$$

A ideia é que o alvo t atraia o caminho. Se t se move, a busca por caminhos mínimos usando A* costuma ser bem mais eficiente que o Dijkstra em casos como esse. (Mostrar vídeo com simulação)

Algoritmo de Dijkstra

Vantagens

Resolve todos os tipos de problemas envolvendo caminhos mínimos

i) 1 – 1 (caminho ótimo s-t): basta parar ao remover t da fila de prioridades

ii) 1 – N (árvore de caminhos ótimos)

iii) N – N (múltiplas árvores): executar Dijkstra $|V|$ vezes, cada uma com uma raiz diferente

Obs: Há um algoritmo que resolve diretamente o subproblema iii)

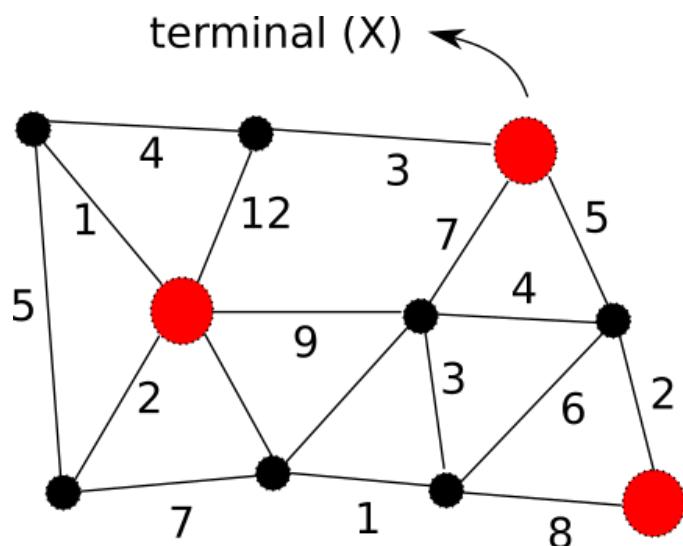
Floyd-Warshall: $O(n^3)$

Steiner Trees (generalização de MST e caminhos mínimos)

O problema da árvore de Steiner

Seja $G = (V, E, w, X, \Lambda)$ onde $w:E \rightarrow R^+$, X é o conjunto de vértices terminais e

$\Lambda = V - X$ é o conjunto dos vértices de Steiner. O objetivo consiste em encontrar o subgrafo de peso mínimo que interconecta todos os vértices terminais (não é necessário usar todos os vértices de Steiner)



Esse problema é NP-Hard Porém dois casos especiais possuem solução ótima:

i) $|X| = 2 \rightarrow$ caminho mínimo (Dijkstra)

ii) $X = V$ ou ($\Lambda = \emptyset$) \rightarrow MST (Prim)

Até o presente momento, os problemas estudados possuem algoritmos que nos oferecem sempre a melhor solução possível, ou seja, soluções ótimas. Em toda instância do problema é garantido que obteremos sempre a melhor solução.

A partir de agora iremos estudar classes de problemas em que isso não se observa. São problemas para os quais muitas vezes teremos aproximações, ou seja, são soluções boas, mas não há garantias de que os algoritmos sempre fornecerão a solução ótima

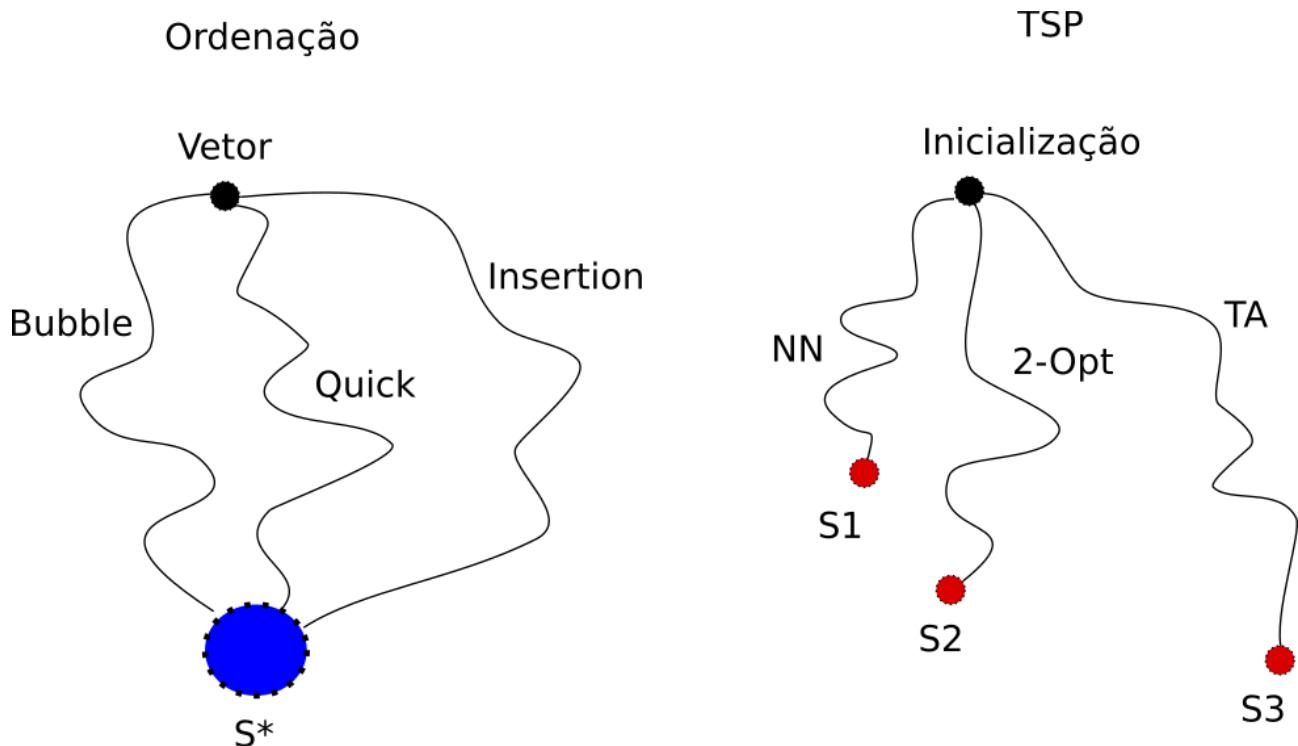
Classificação de Problemas

Problemas
- ordenação
- busca
- TSP

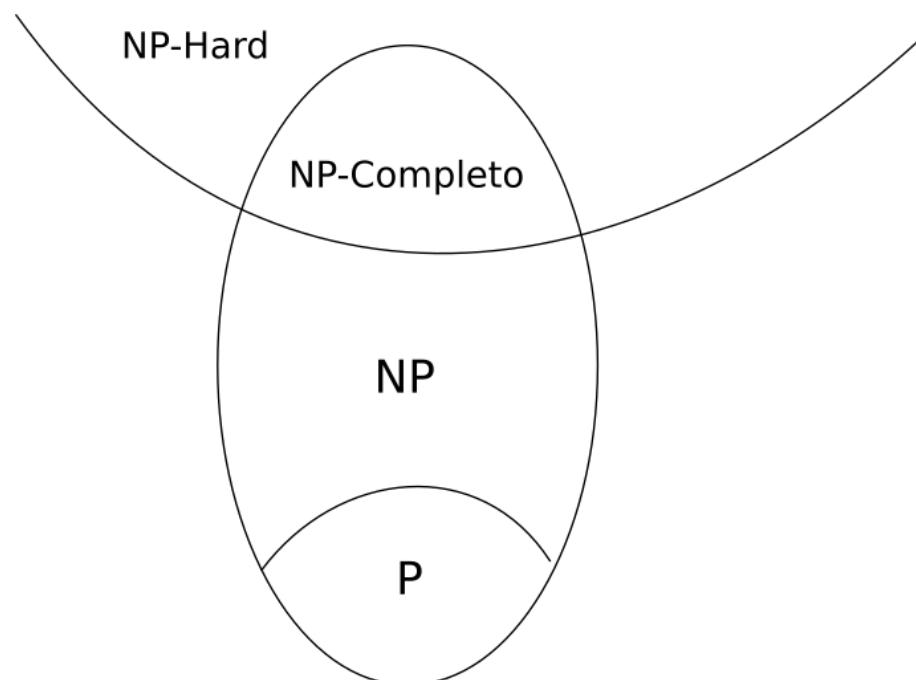
X

Algoritmos
- Bubble, Quick, Insertion, ... ($O(n^2)$, $O(n \log n)$, ...)
- Sequencial, Binária, Largura, ,...
- NN, 2-Opt, Twice-Around,...

O que ocorre é que em alguns problemas todos os algoritmos chegam no mesmo ponto, independente da inicialização e do caminhos. Em problemas mais complexos isso não ocorre.



Classes de Problemas



- i) $p \in P$ se \exists algoritmo A $O(n^k)$ (polinomial) que resolve p
- ii) $p \in NP$ se \nexists algoritmo A $O(n^k)$ que resolve p (mas dada uma solução é fácil verificar se ela é válida, em tempo rápido descobre-se)
- iii) $\exists p \notin NP$ para os quais até mesmo verificar se uma dada solução é válida sequer é viável: NP-Hard

Porque problemas NP-Completos são tão importantes e estudados em computação?

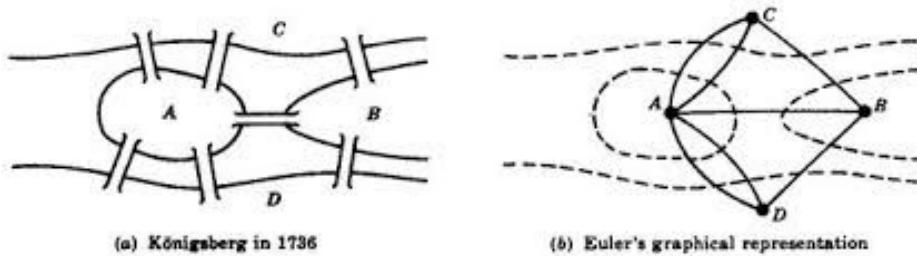
Ex: TSP

Devido a possibilidade de que se for descoberto um algoritmo polinomial para um problema NP-Completo, todos os problemas em NP podem ser resolvidos de maneira eficiente: $P = NP ?$

Grafos Eulerianos

Trata-se de uma classe muito importante de grafos que modelam diversos problemas reais.

O estudo de grafos Eulerianos nos remete as origens da Teoria dos Grafos, quando em 1736, Leonard Euler propôs o problema das 7 pontes de Königsberg



Def: Trilha de Euler

É uma trilha que engloba toda aresta de G (ou seja, passa exatamente 1 única vez em cada aresta)
Pode ter origem e destino diferentes

Def: Tour de Euler/Círculo Euleriano

É toda trilha de Euler fechada

Def: $G = (V, E)$ é Euleriano \Leftrightarrow G admite um tour de Euler

Teorema (Euler, 1976):

Seja $G = (V, E)$ um grafo conexo. G é Euleriano se e somente se satisfaz a propriedade E a seguir:

$$\forall v \in V (d(v) \bmod 2 = 0)$$

ou seja, se o grau de todo vértice é par.

PROVA: (ida) G é Euleriano \rightarrow Possui propriedade E

1. G admite um tour de Euler $W = u a b c d \dots u$

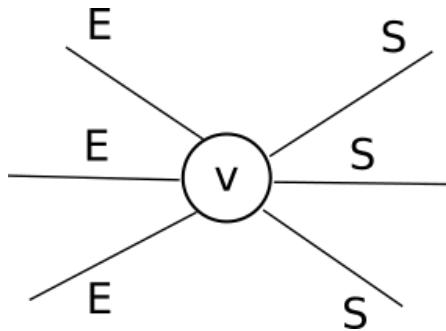
$$\forall v \in V \text{ aparece em } W \text{ (com repetições)}$$

$$\forall e \in E \text{ aparece em } W \text{ (exatamente uma única vez)}$$

2. Selecione o vértice inicial u

Como ele só aparece no inicio e fim, então $d(u) = 2$ (par)

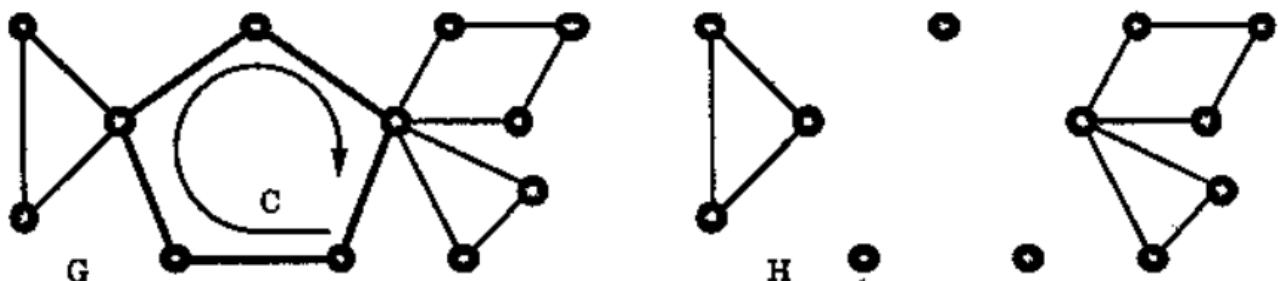
3. Escolha os vértices intermediários $v \neq u$



Como o tour não termina em v , segue que se consigo entrar em v , deve haver uma saída, ou seja, para cada entrada existe uma saída correspondente, o que significa que o grau do vértice é um número do tipo $2n$ (par)

(volta) G possui propriedade $E \rightarrow G$ é Euleriano

1. Como G é conexo, $\nexists v \in V$ tal que $d(v) = 0$
2. $\forall v \in V d(v) \geq 2$ então $\bar{d} \geq 2$ o que implica na existência de ao menos 1 ciclo C em G
3. Se o ciclo C contém toda aresta de G , então G é Euleriano.
4. Caso contrário, faça $H = G - C$. Note que os graus dos vértices de H são todos pares. Então cada componente de H é Euleriano.



5. Sendo assim, pode-se combinar esses ciclos arestas disjuntos para formar um tour de Euler: inicie num vértice v de C em G e percorra as arestas uma a uma, deletando-as até encontrar um vértice de junção (pertence a mais de 1 ciclo e por isso possui grau maior que 2). Siga percorrendo e deletando todas as arestas desse novo ciclo C' até voltar a C ou encontrar um novo ciclo C'' (vértice de grau maior que 2). Repita esse processo até que não restem mais arestas em H .

6. Como é possível extrair um tour de Euler de G , segue que G é Euleriano

Esse resultado é importante pois nos fornece uma maneira eficiente para decidir se um dado grafo G é Euleriano ou não. Decidir se G é Euleriano é algo trivial, feito em tempo polinomial (basta verificar se a lista de graus contém apenas números pares)

Veremos a seguir um algoritmo para a construção de um tour de Euler válido a partir de um grafo G Euleriano.

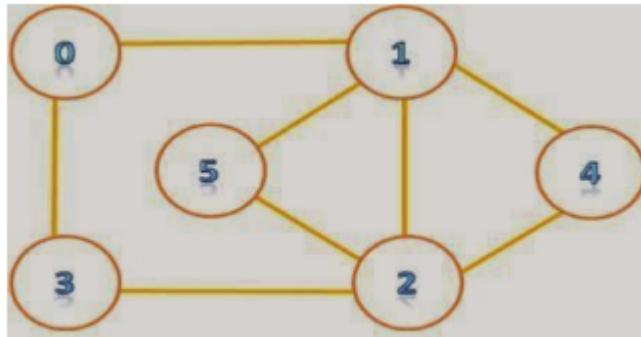
Algoritmo de Fleury

Entrada: $G = (V, E)$ Euleriano

Saída: Tour de Euler

1. Escolha um vértice inicial v_0 qualquer
2. Se $W_i = v_0 e_1 v_1 e_2 v_2 \dots e_i v_i$ escolha e_{i+1} tal que
 - e_{i+1} é incidente a v_i
 - e_{i+1} não é ponte no subgrafo $G - W_i$ (a menos que seja a única opção)
3. Pare se W_i tiver $\forall e \in E$. Senão, volta para o passo 2.

Ex:



Variáveis:

E^- : conjunto das arestas ponte com extremidade no vértice atual v

E^+ : conjunto de arestas não ponte com extremidade no vértice atual v

$T^{(i)}$: tour de Euler no i -ésimo passo

$T^{(0)} = [0]$ (iniciaremos o tour no vértice 0)

i	E^-	E^+	$T^{(i)}$
1	\emptyset	$\{(0,1), (0,3)\}$	$[0, 1]$
2	\emptyset	$\{(1,2), (1,4), (1,5)\}$	$[0, 1, 2]$
3	$\{(2,3)\}$	$\{(2,4), (2,5)\}$	$[0, 1, 2, 4]$
4	$\{(4,1)\}$	\emptyset	$[0, 1, 2, 4, 1]$
5	$\{(1,5)\}$	\emptyset	$[0, 1, 2, 4, 1, 5]$
6	$\{(5,2)\}$	\emptyset	$[0, 1, 2, 4, 1, 5, 2]$
7	$\{(2,3)\}$	\emptyset	$[0, 1, 2, 4, 1, 5, 2, 3]$
8	$\{(3,0)\}$	\emptyset	$[0, 1, 2, 4, 1, 5, 2, 3, 0]$

Problema: O algoritmo de Fleury é mais complicado de se implementar pois depende de quanto bem você consegue detectar pontes em grafos G. Há vários algoritmos para esse fim, sendo um dos mais conhecidos o método de Tarjan (Tarjan bridge finding algorithm). Outros métodos baseados na busca em profundidade (DFS) também existem.

Veremos a seguir um outro algoritmo para construir um tour de Euler que não depende de detecção de ciclos.

Algoritmo de Hierholzer

Entrada: G Euleriano

Saída: Tour de Euler T

1. Escolha um vértice inicial v e insira o na pilha S. Todas as arestas iniciam desmarcadas.

2. Enquanto S não é vazia

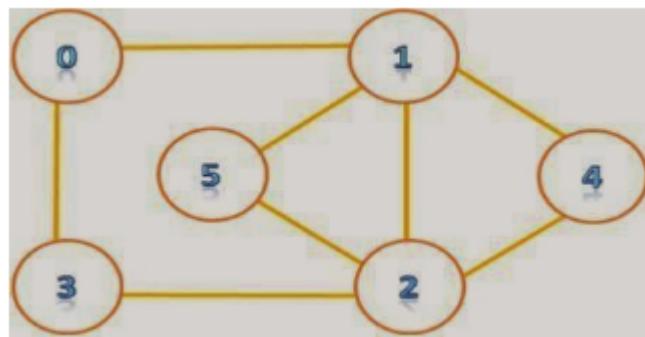
a. Seja u o vértice do topo.

b. Se u possui uma aresta incidente desmarcada para um vértice w, então adicione w a pilha S e marque aresta (u,w)

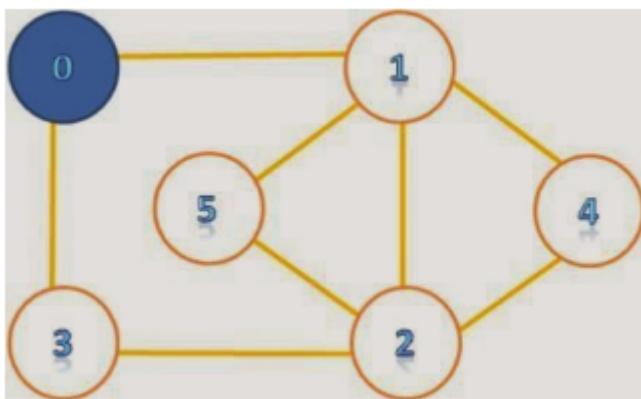
c. Se u não possui aresta incidente desmarcada, remova u do topo da pila S e imprima u.

3. Quando S se tornar vazia, o algoritmo terá imprimido uma sequencia de vértices T que corresponde a um tour de Euler.

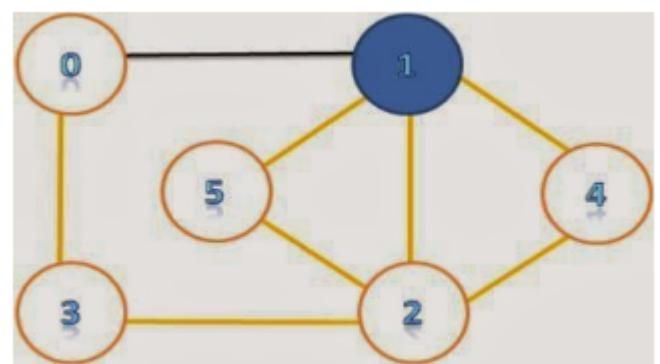
Ex:



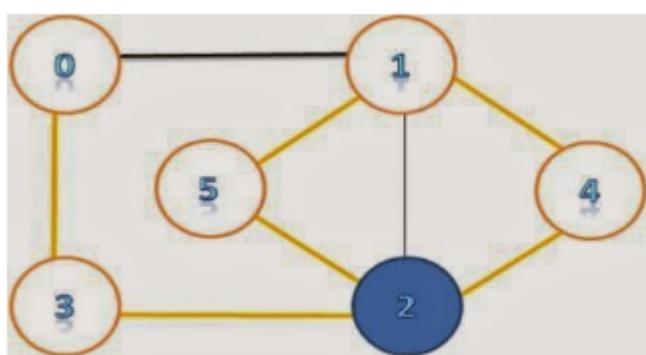
$$S = \emptyset, \quad T = \emptyset$$



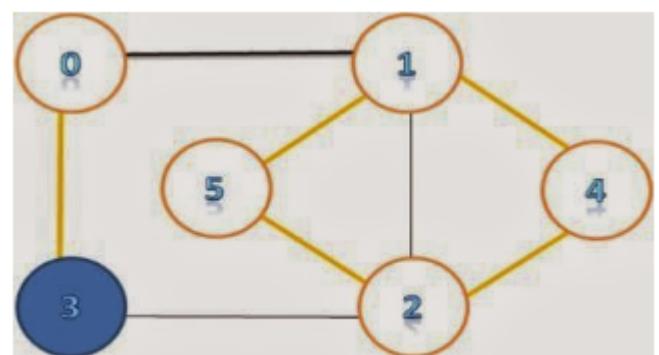
$$S = [0], \quad T = \emptyset$$



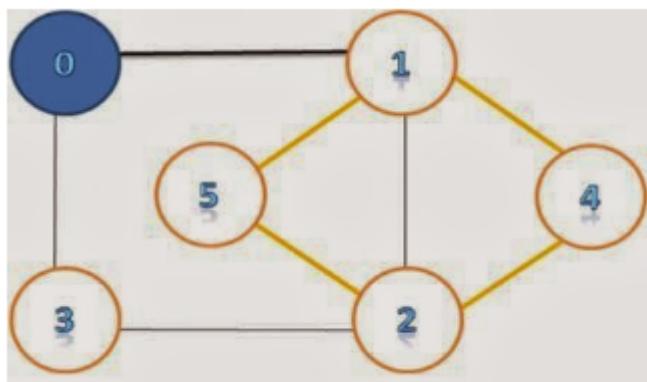
$$S = [0, 1], \quad T = \emptyset$$



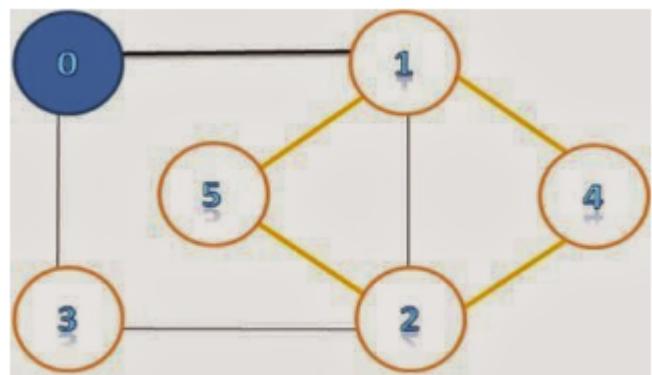
$$S = [0, 1, 2], \quad T = \emptyset$$



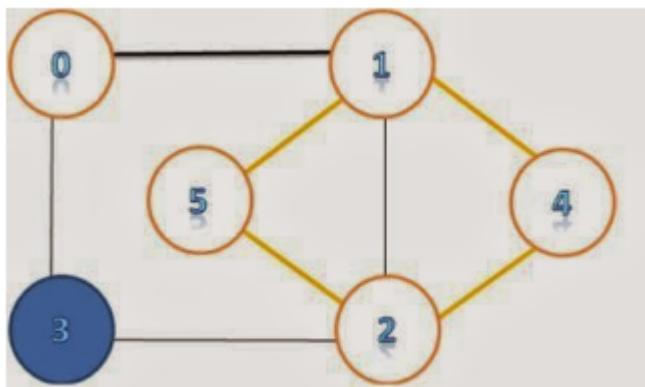
$$S = [0, 1, 2, 3], \quad T = \emptyset$$



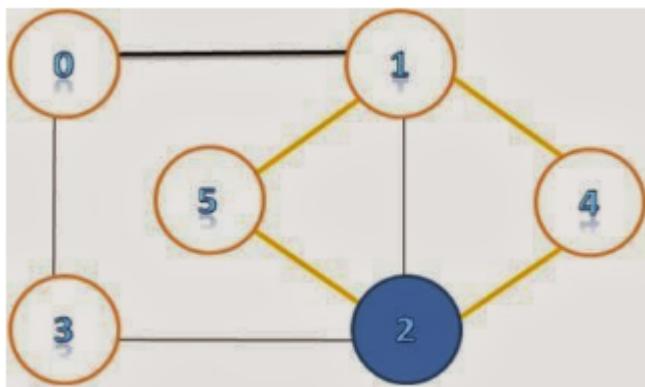
$S = [0, 1, 2, 3, 0]$, $T = \emptyset$



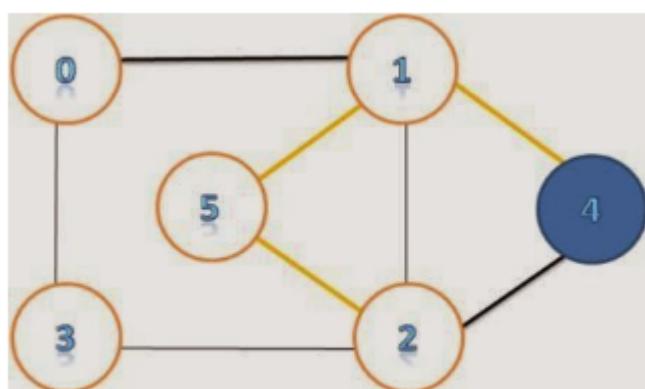
$S = [0, 1, 2, 3]$, $T = [0]$



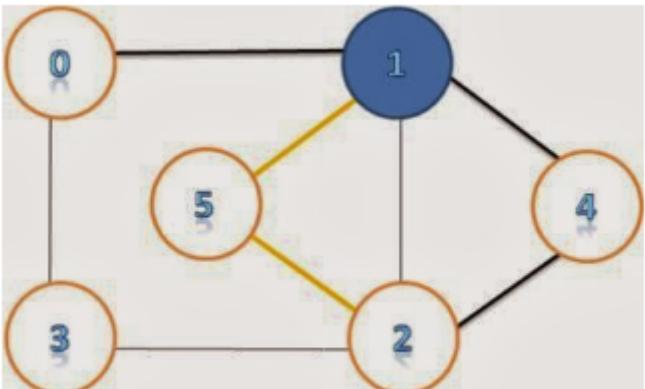
$S = [0, 1, 2]$, $T = [0, 3]$



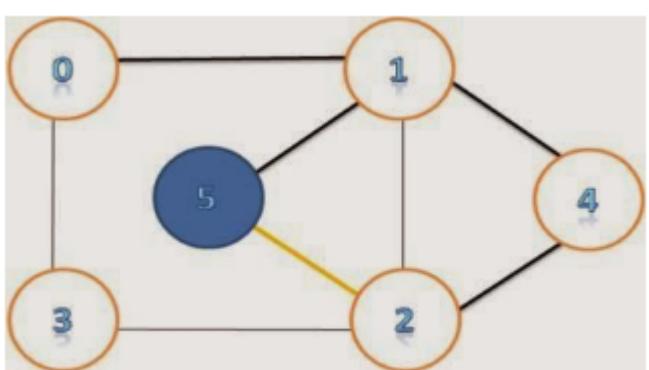
Há arestas incidentes a 2, continua em S



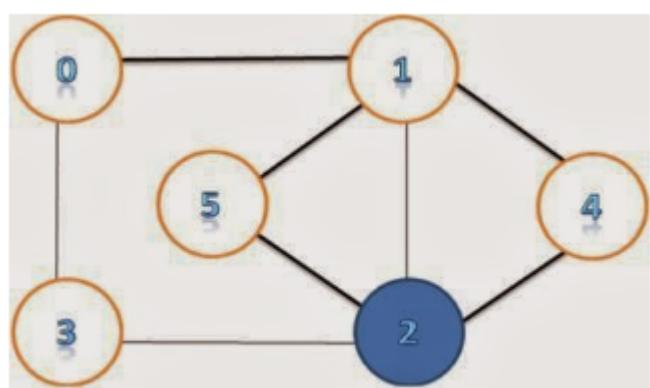
$S = [0, 1, 2, 4]$, $T = [0, 3]$



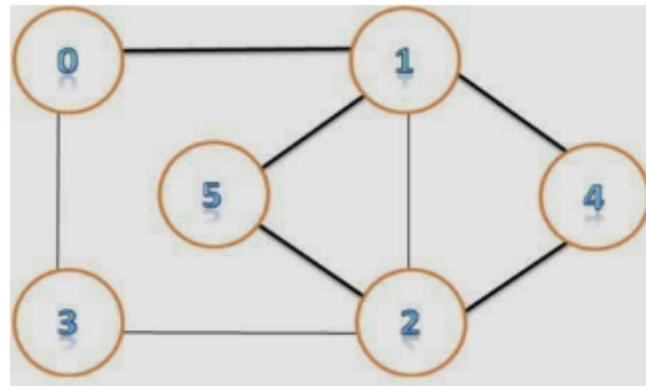
$S = [0, 1, 2, 4, 1]$, $T = [0, 3]$



$S = [0, 1, 2, 4, 1, 5]$, $T = [0, 3]$



$S = [0, 1, 2, 4, 1, 5, 2]$, $T = [0, 3]$



$$S = \emptyset, \quad T = [0, 3, 2, 5, 1, 4, 2, 1, 0]$$

Veremos a seguir um importante problema envolvendo grafos Eulerianos.

O Problema do Carteiro Chinês (Chinese Postman Problem)

Objetivo: encontrar um tour de Euler de peso mínimo num grafo G ponderado

Solução trivial: G é Euleriano

Algoritmo de Fleury

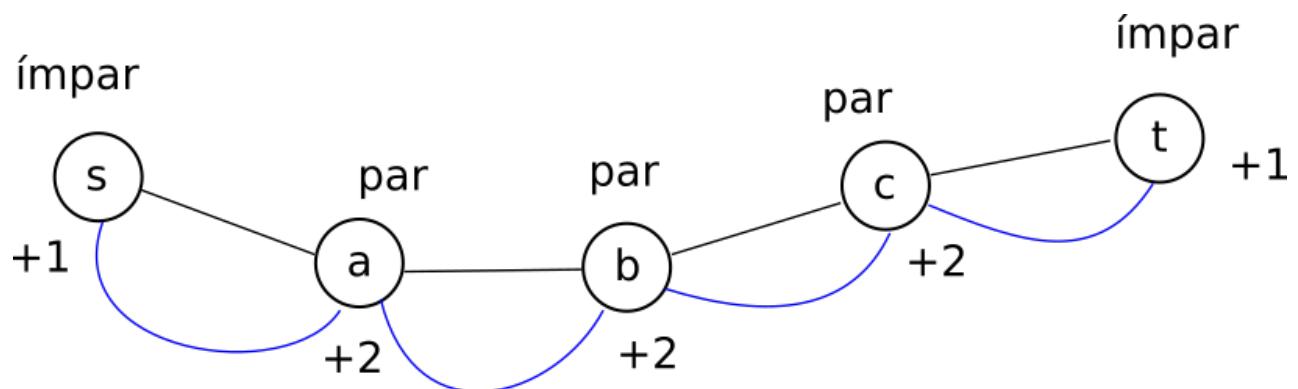
Problema: E se o grafo G não for Euleriano?

i) Transformar G num supergrafo Euleriano G^*

Objetivo: Encontrar, por duplicação de arestas, um supergrafo G^* de modo a minimizar

$$\sum_{e \in E(G^*) - E(G)} w(e) \quad (\text{minimizar soma dos pesos das arestas duplicadas}) \\ (\text{conj. das arestas novas})$$

Solução: Por simplificação iremos considerar o caso mais simples do problema: o grafo G não é Euleriano pois existem exatamente 2 vértices de grau ímpar.



Aplicar algoritmo de Dijkstra para encontrar caminho mínimo P_{st}^* entre os vértices ímpares.

ii) Aplicar algoritmo de Fleury em G^* para obter um tour de Euleriano

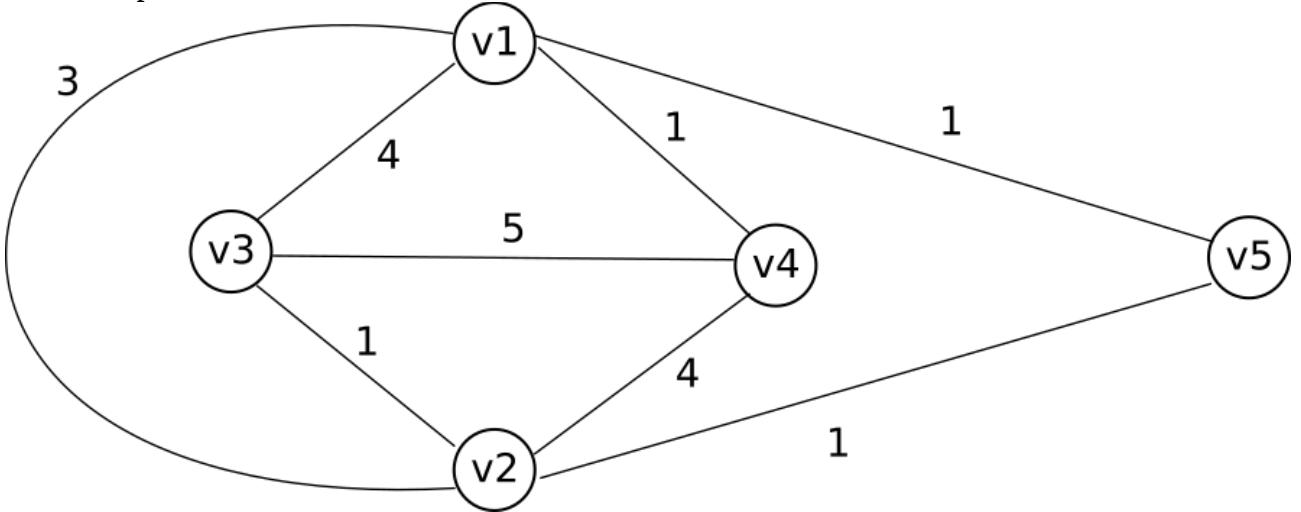
Obs: 1) Note que o passo crítico na solução é i), uma vez que ao realizar a duplicação de maneira incorreta, todo tour de Euler subsequente não é uma solução válida.

2) O caso geral do problema não limita o número de vértices ímpares. Nesse caso a solução é muito mais complexa, envolvendo subproblemas como encontrar um emparelhamento mínimo

a) Encontrar caminhos mínimos entre todos os pares de vértices ímpares

- b) Supondo n vértices ímpares, montar um grafo K_n em que o peso das arestas é o peso de cada caminho.
c) Encontrar um emparelhamento mínimo no grafo completo em questão

Ex: Supor que v_1 é o correio central. Carteiro deve sair de v_1 entregar cartas e voltar a v_1 andando o mínimo possível



1) Transformar G em G^*

Aplicar algoritmo de Dijkstra de v_3 a v_4

Fila

	v_1	v_2	v_3	v_4	v_5	
$\lambda^{(0)}(v)$	∞	∞	0	∞	∞	
$\lambda^{(1)}(v)$	4	1		5		
$\lambda^{(2)}(v)$	4			5	2	
$\lambda^{(3)}(v)$	3			5		
$\lambda^{(4)}(v)$				4		

Ordem de acesso aos vértices

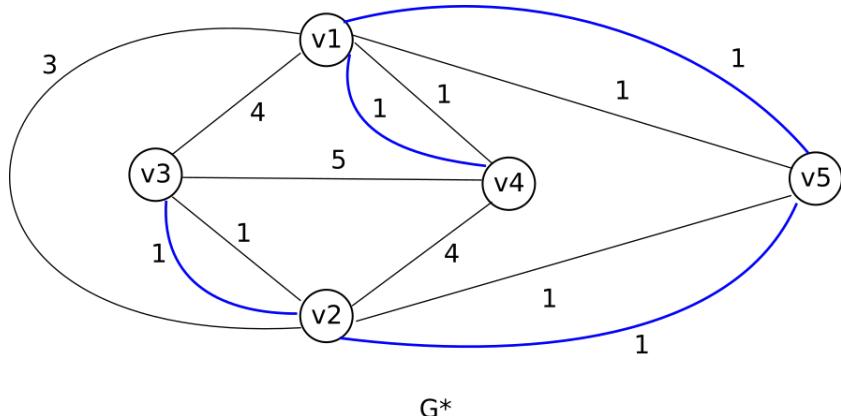
u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
v_3	$\{v_1, v_2, v_4\}$	$\lambda(v_1) = \min\{\infty, 4\} = 4$ $\lambda(v_2) = \min\{\infty, 1\} = 1$ $\lambda(v_4) = \min\{\infty, 5\} = 5$	$\pi(v_1) = v_3$ $\pi(v_2) = v_3$ $\pi(v_4) = v_3$
v_2	$\{v_1, v_4, v_5\}$	$\lambda(v_1) = \min\{4, 1+3\} = 4$ $\lambda(v_4) = \min\{5, 1+4\} = 5$ $\lambda(v_5) = \min\{\infty, 1+1\} = 2$	---
v_5	$\{v_1\}$	$\lambda(v_1) = \min\{4, 2+1\} = 3$	$\pi(v_1) = v_5$
v_1	$\{v_4\}$	$\lambda(v_4) = \min\{5, 3+1\} = 4$	$\pi(v_4) = v_1$
v_4	\emptyset	---	---

Mapa de predecessores (árvore final)

v	v1	v2	v3	v4	v5	
$\pi(v)$	v5	v3	---	v1	v2	

Portanto, pelo mapa de predecessores, o caminho mínimo entre v3 e v4 é dado por:
 $P^* = v3 \rightarrow v2 \rightarrow v5 \rightarrow v1 \rightarrow v4$

Assim, o supergrafo G^* fica:



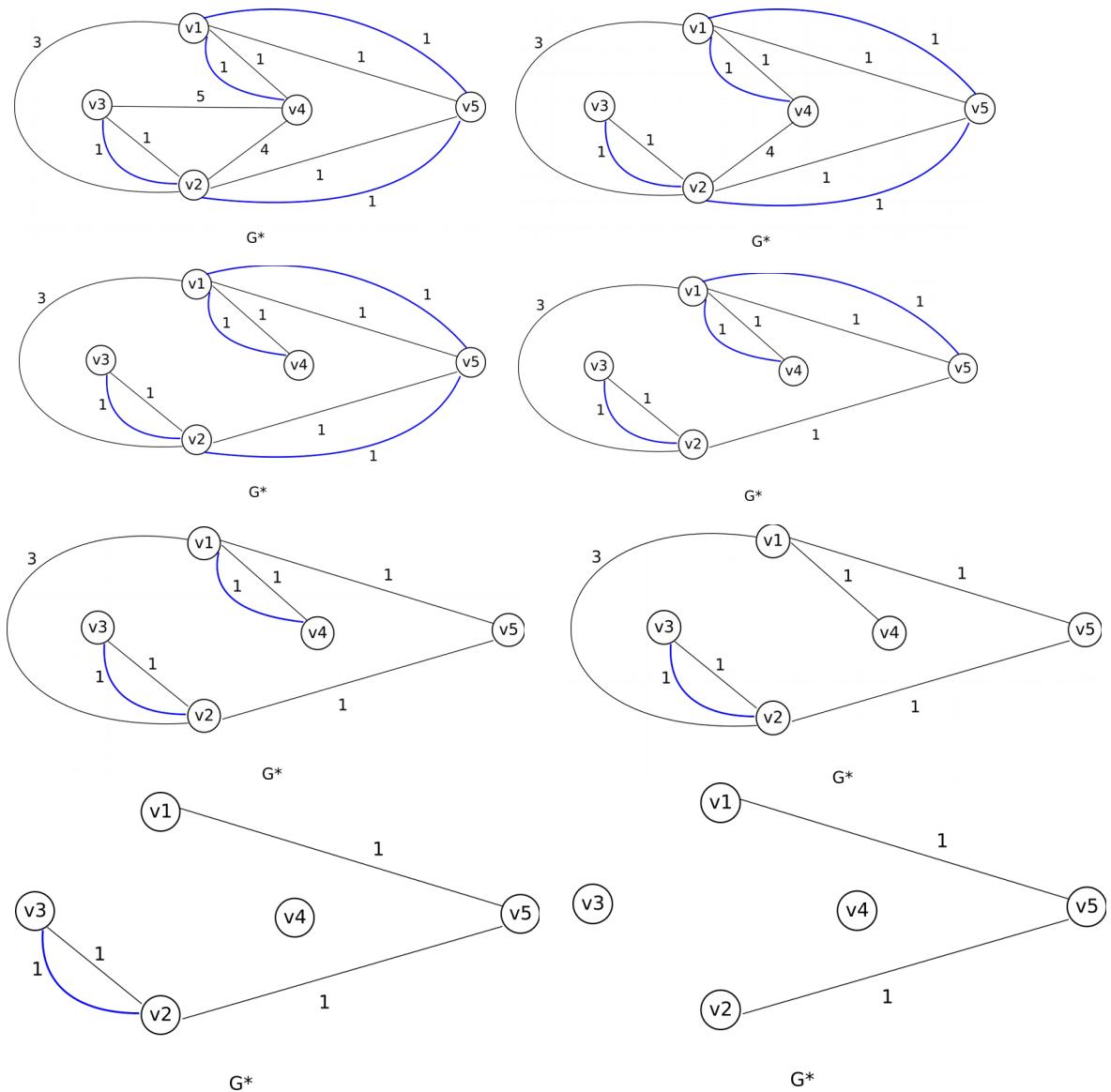
2) Obter o tour de Euler com o algoritmo de Fleury ou Hierholzer

Na verdade, qualquer tour de Euler válido será de peso mínimo agora. Existem inúmeras possibilidades, dentre elas a solução mostrada a seguir.

$$T^{(0)} = [v_1] \quad (\text{iniciaremos o tour no vértice } 1)$$

i	E^-	E^+	$T^{(i)}$
1	\emptyset	$\{(v1,v2), (v1,v3), (v1,v4), (v1,v4), (v1,v5)\}$	v3
2	\emptyset	$\{(v3,v2), (v3,v2), (v3,v4)\}$	v4
3	\emptyset	$\{(v4,v1), (v4,v1), (v4,v2)\}$	v2
4	\emptyset	$\{(v2,v1), (v2,v3), (v2,v3), (v2,v5), (v2,v5)\}$	v5
5	\emptyset	$\{(v5,v2), (v5,v1), (v5,v1)\}$	v1
6	\emptyset	$\{(v1,v2), (v1,v4), (v1,v4), (v1,v5)\}$	v4
7	$\{(v4,v1)\}$	\emptyset	v1
8	\emptyset	$\{(v1,v2), (v1,v5)\}$	v2
9	$\{(v2,v5)\}$	$\{(v2,v3), (v2,v3)\}$	v3
10	$\{(v3,v2)\}$	\emptyset	v2
11	$\{(v2,v5)\}$	\emptyset	v5
12	$\{(v5,v1)\}$	\emptyset	v1

$T = v1\ v3\ v4\ v2\ v5\ v1\ v4\ v1\ v2\ v3\ v2\ v5\ v1$



Obs: Cabe ressaltar que num grafo com muitos vértices, há um número imenso de possíveis tours de Euler, mas também há vários pontos em que somos obrigados a evitar pontes para o algoritmo não falhar

Grafos Hamiltonianos

Relação direta com o problema do caixeiro viajante (Travelling Salesman Problem – TSP), que é NP-Completo

Def: Um ciclo Hamiltoniano é um ciclo que engloba todo vértice de G (ciclo não permite repetição)

Def: Um grafo G é Hamiltoniano se e somente se G possui um ciclo Hamiltoniano (dual de Euler)

Obs: $\forall n > 2$ K_n é Hamiltoniano

Processo de crescimento das arestas

Supor um grafo G arbitrário não Hamiltoniano

$$G \rightarrow G' \rightarrow G'' \rightarrow G''' \rightarrow \dots \rightarrow K_n$$

$$+e \quad +e \quad +e \quad +e \quad +e$$

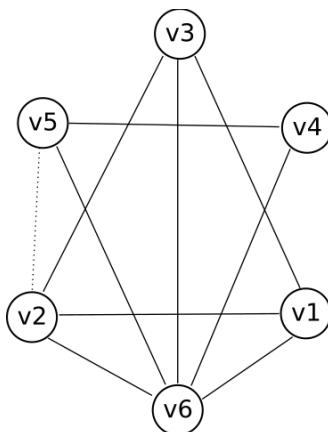
A cada passo, uma aresta é inserida em G. Como K_n é Hamiltoniano, segue que em algum momento entre G e K_n , o grafo torna-se Hamiltoniano. Usaremos essa noção para a definição a seguir.

Def: Um grafo G é não Hamiltoniano maximal se G não é Hamiltoniano mas a adição de qualquer nova aresta o torna Hamiltoniano (está na iminência de se tornar Hamiltoniano)

Ex: considere o grafo G a seguir. Note que G não admite um ciclo Hamiltoniano mas ao adicionar qualquer aresta nova, G torna-se Hamiltoniano

No caso da aresta (v2, v5) temos: $v_6 - v_4 - v_5 - v_2 - v_1 - v_3 - v_6$

No caso da aresta (v1, v4) temos: $v_6 - v_5 - v_4 - v_1 - v_2 - v_3 - v_6$



Pergunta: Como decidir se um dado grafo G é Hamiltoniano? Problema difícil. (NP-Completo)
Porque? Melhor resultado que se tem até hoje

Teorema (Dirac, 1952)

Seja $G = (V, E)$ um grafo básico simples com $|V| = n > 2$. Então,

$$\forall v \in V (d(v) \geq \frac{n}{2}) \rightarrow G \text{ é Hamiltoniano}$$

propriedade H

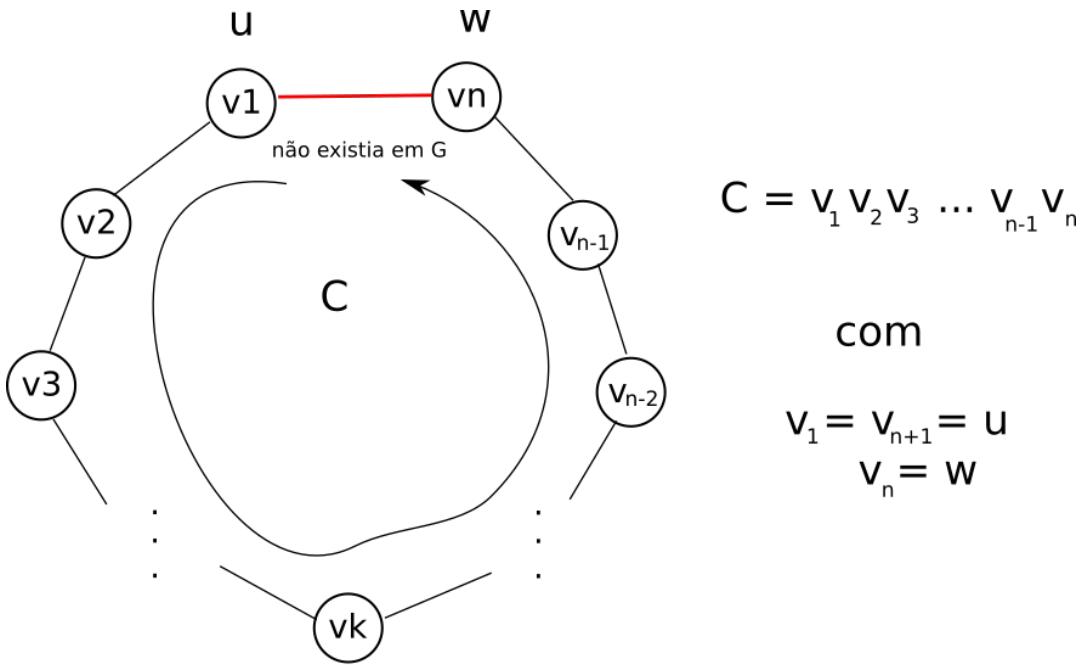
Em outras palavras, o que esse resultado nos diz é que se cada um dos vértices de G se liga pelo menos a metade dos demais, então G é Hamiltoniano. (esse resultado ainda é considerado o estado da arte na identificação de grafos Hamiltonianos, no sentido que não há nada mais poderoso)

PROVA: (por contradição) Ideia é verificar que não pode existir G que satisfaça propriedade H mas não seja Hamiltoniano

Suponha que $\exists G = (V, E)$ básico simples que satisfaz propriedade mas não é Hamiltoniano. Considere G como sendo não Hamiltoniano maximal. Então \exists em G $u, v \in V$ não adjacentes.

Faça $H = G + (u, v)$. H é Hamiltoniano, então \exists ciclo C que passa por $\forall v \in V$

Chamaremos $u = v_1$ e $v = v_n$



$$H = G + (u, w)$$

Defina 2 conjuntos de vértices, S e T, como segue:

$$S = \{ v_i : \exists \text{ aresta que liga } u \text{ a } v_{i+1} \}$$

$$T = \{ v_j : \exists \text{ aresta que liga } w \text{ a } v_j \}$$

Obs: Quem está em S? Não importa! Quantos elementos estão em S? Isso importa

Assim, temos que $|T| = d(w)$ e $|S| = d(u)$ (número de ligações).

Agora, iremos verificar que \exists ao menos um vértice que não está em S nem em T: v_n

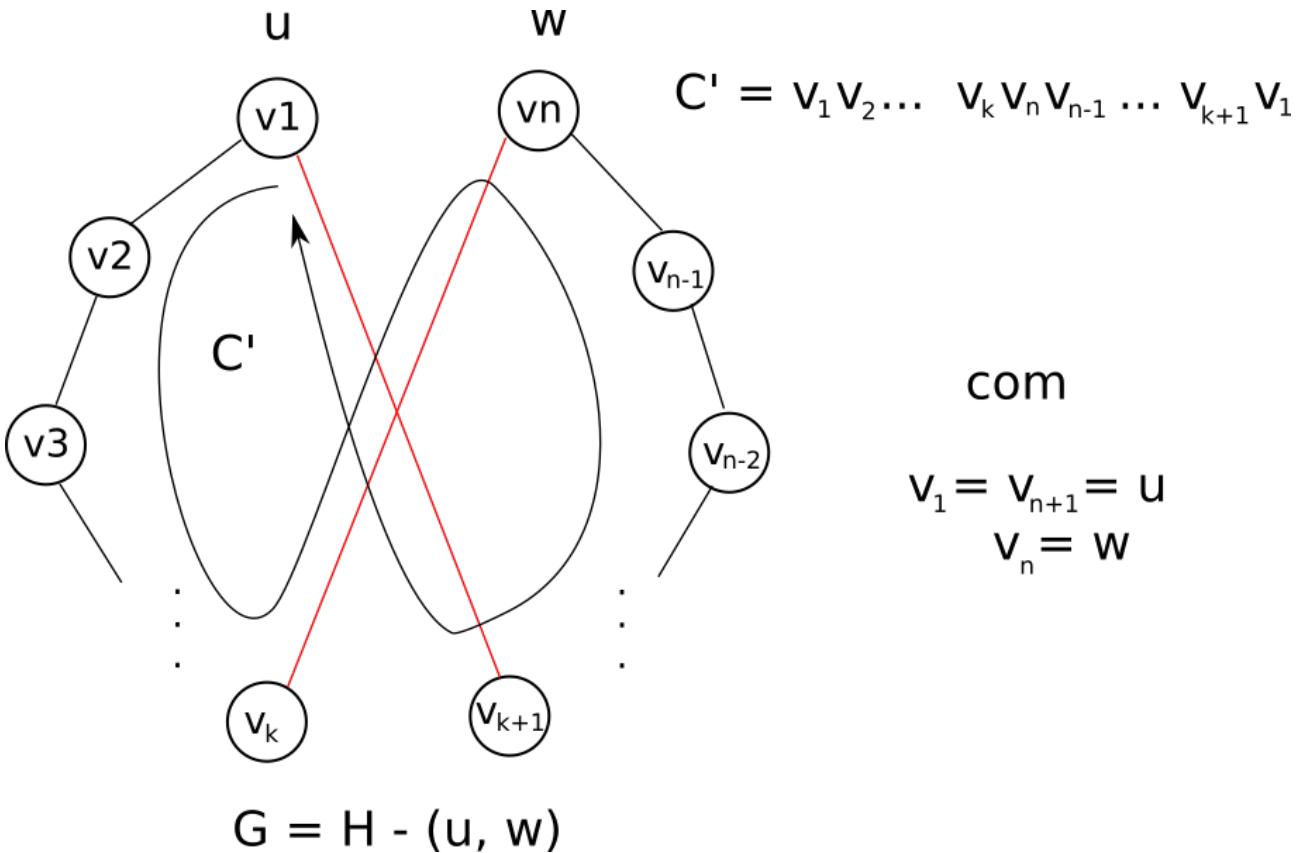
Note que:

- i) $v_n \notin S$: Porque? \exists aresta que liga u a $v_{n+1} = u$? Não pois não há loops!
- ii) $v_n \notin T$: Porque? \exists aresta que liga w a $v_n = w$? Não pois não há loops!

Dessa forma, $v_n \notin S \cup T$, o que implica que $|S \cup T| < n$

O próximo passo consiste em responder a pergunta: $\exists v_k \in S \cap T$? Suponha que sim, se chegarmos numa contradição é porque não pode existir. Vamos considerar que $v_k \in S \cap T$. Então,

- i) se $v_k \in S$: \exists aresta que liga u a v_{k+1}
- ii) se $v_k \in T$: \exists aresta que liga w a v_{k+1}



Porém, nesse caso G seria Hamiltoniano pois existiria um ciclo C' que envolve todo vértice de G . Isso fere a definição inicial de que G é não Hamiltoniano Maximal, gerando uma contradição e portanto invalidando a suposição de que $\exists v_k \in S \cap T$. O fato é que $\nexists v_k \in S \cap T$. Isso implica que $S \cap T = \emptyset$.

Dessa forma, temos:

$$|S \cup T| = |S| + |T| - |S \cap T| = d(u) + d(w)$$

Mas como de (*) temos que $|S \cup T| < n$ finalmente chegamos a:

$$d(u) + d(w) < n$$

o que é uma contradição para a propriedade H , pois todo grafo que satisfaz H tem

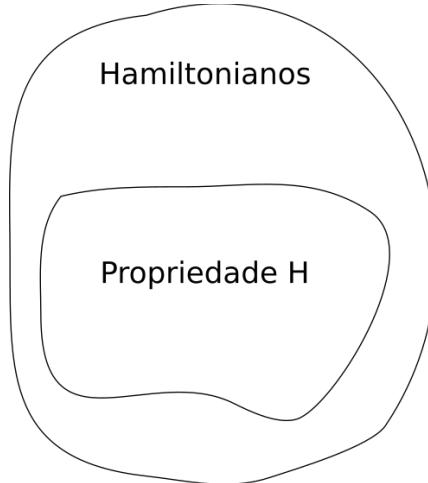
$$d(u) \geq \frac{n}{2} \quad \text{e} \quad d(v) \geq \frac{n}{2}$$

para quaisquer u e w em V , ou seja, $d(u) + d(w) \geq n$

Conclusão: Não existe grafo G para os quais a propriedade H seja válida e G não seja Hamiltoniano. (se H é verdade então é certeza que G é Hamiltoniano)

Mas isso gerante que é simples decidir se um dado G é Hamiltoniano? Não, pois posso ter grafos G para os quais H falha e mesmo assim eles são Hamiltonianos. Exemplo: grafo 2-regular conexo

Problema em aberto: ninguém nunca demonstrou a volta. Nem mesmo propôs um critério mais efetivo (do tipo se e somente se)



O Problema do Caixeiro Viajante (Travelling Salesman Problem – TSP)

Objetivo: Dado um grafo $G = (V, E, w)$ encontrar um ciclo Hamiltoniano de custo mínimo, ou seja, obter o ciclo C que minimiza:

$$w(C) = \sum_{e \in C} w(e)$$

Descrição do problema: A partir de um vértice inicial v_0 , visitar todos os demais uma única vez, voltando para v_0 , caminhando o mínimo possível

Ex: Motoboy deve sair da lanchonete, passar por 5 casas e voltar andando o mínimo possível

Relevância:

- Modelar e resolver vários problemas em diversas áreas da ciência (logística, seq. genoma, NASA)
- NP-Completo

Desafios: em qualquer instância do TSP, precisamos de respostas para 2 perguntas:

- i) $G = (V, E)$ é Hamiltoniano? (existe ciclo Hamiltoniano)
- ii) Se sim, como obter ciclo Hamiltoniano de custo mínimo?

Limitações e restrições

a) Em relação a i)

Na prática, assume-se uma hipótese simplificadora: grafo de entrada é completo (K_n)

Heurística para converter $G = (V, E)$ em K_n

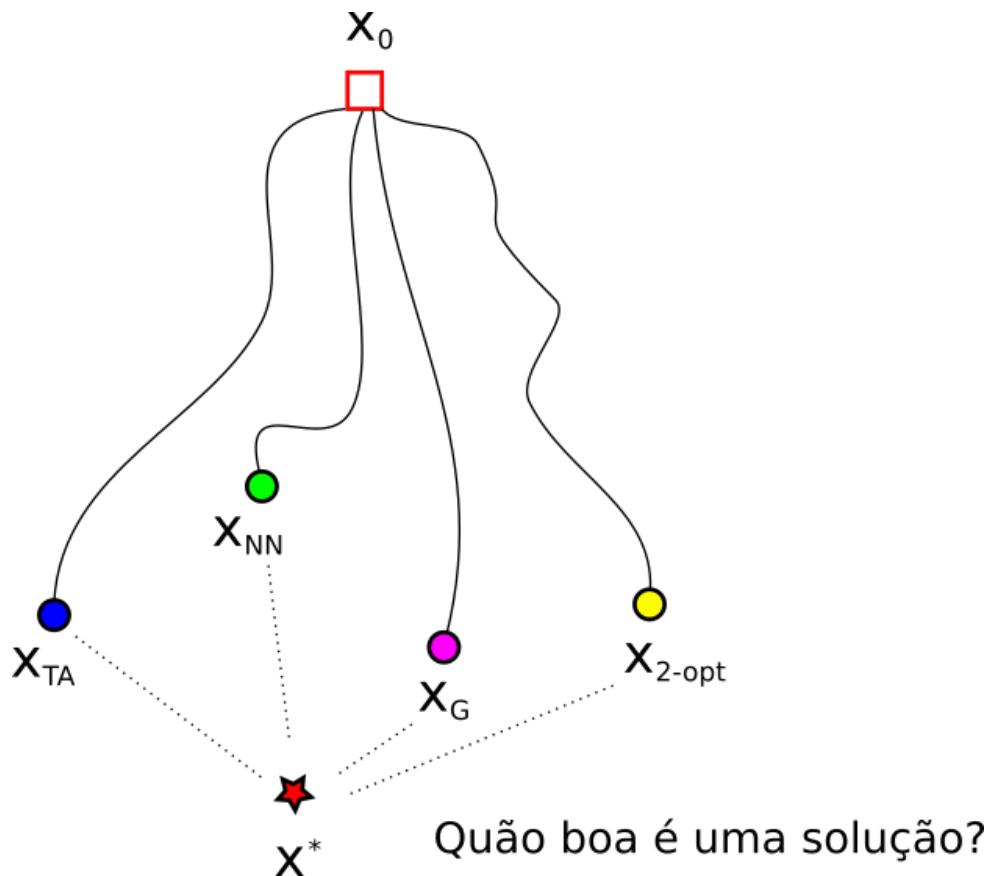
- se $e \in E$, mantém $w(e)$ intacto
- se $e \notin E$, $w(e) = \sum_{e \in E} w(e)$ (soma de todos os pesos)

b) Em relação a ii)

Não se conhece algoritmo ótimo (que produz sempre a melhor solução)

Não há garantias de que termos a melhor solução x^*

Aceitamos algoritmos subótimos (aproximações)



Uma pergunta que deseja-se responder ao se utilizar algoritmos sub-ótimos é: quão próximo da solução ótima estamos com aquele determinado algoritmo?

Definem-se

- $f(x^*)$: custo da solução ótima (melhor possível)
- $f(x_A)$: custo da solução obtida pelo algoritmo A

Dizemos que o algoritmo A fornece uma c -aproximação para o TSP se $f(x_A) \leq c f(x^*)$

Def: Grafo Euclidiano

$G = (V, E)$ é Euclidiano se seus vértices representam pontos no plano e os pesos das arestas são distâncias entre os pontos.

Comentário: Essencialmente o que se faz com essa definição é considerar um grafo como um objeto contínuo, ou seja, se estamos no meio da aresta (u,v) sabemos exatamente o quão próximo estamos de u ou v . Isso representa bem cidades e estradas, bem como outros cenários do mundo real.

Algoritmos para TSP

Por motivos de simplificação consideraremos que todos os grafos são Hamiltonianos (se não forem, podem ser facilmente convertidos para um usando a heurística descrita anteriormente)

Algoritmo 1: Nearest Neighbor (NN)

Ler grafo $G = (V, E, w)$

$H \leftarrow \{v_0\}$ (ponto de partida)

Enquanto $|H| < n$ {

 Encontrar v_i mais próximo do último elemento de H

$H \leftarrow H \cup \{v_i\}$

}

$H \leftarrow H \cup \{v_0\}$

Prop: A solução obtida pelo algoritmo NN em grafos Euclidianos satisfaz:

$$f(x_{NN}) \leq \frac{1}{2}(\log_2 n + 1)f(x^*) \quad (\text{se grafo cresce a qualidade da solução cai})$$

Algoritmo 2: Twice-Around

Ler grafo $G = (V, E, w)$

$H \leftarrow \emptyset$

Passo 1. $T \leftarrow \text{MST}(G)$ (Kruskal ou Prim)

Para cada $e \in T$

$T \leftarrow T + e$ (Duplique aresta e)

Passo 2. Encontre um circuito Euleriano L em T (Fleury)

Passo 3. Enquanto $L \neq \emptyset$ {

Escolha sequencialmente $l_k \in L$

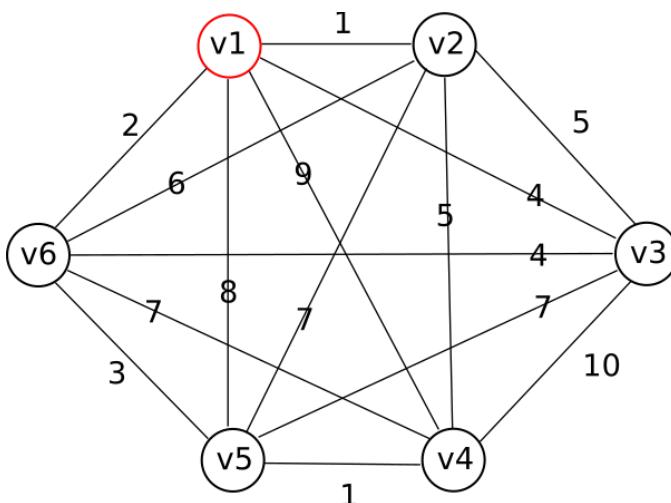
Se $l_k \notin H$ então

$H \leftarrow H \cup \{l_k\}$

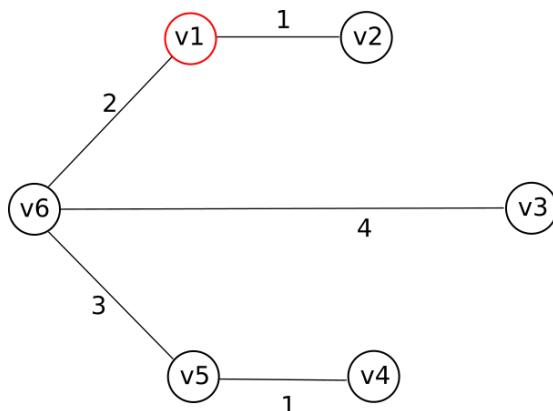
$L \leftarrow L - \{l_k\}$

Obs: Há uma propriedade importante que relaciona árvores e grafos Eulerianos.

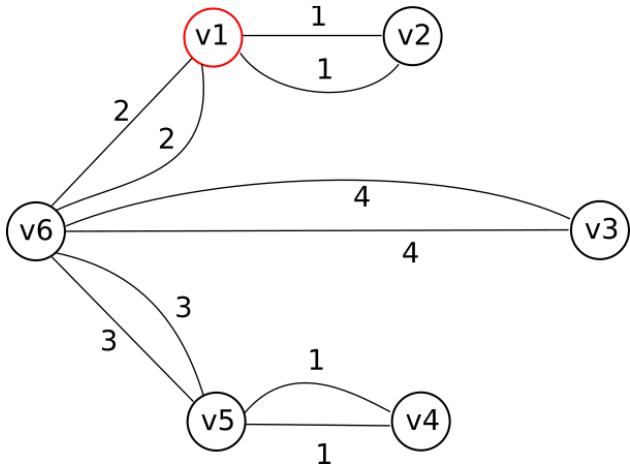
Se duplicarmos a aresta de uma árvore, o grafo resultante é Euleriano



Passo 1: MST a partir de G (Aplicando Kruskal ou Prim)



Duplicando as arestas, temos um grafo Euleriano:



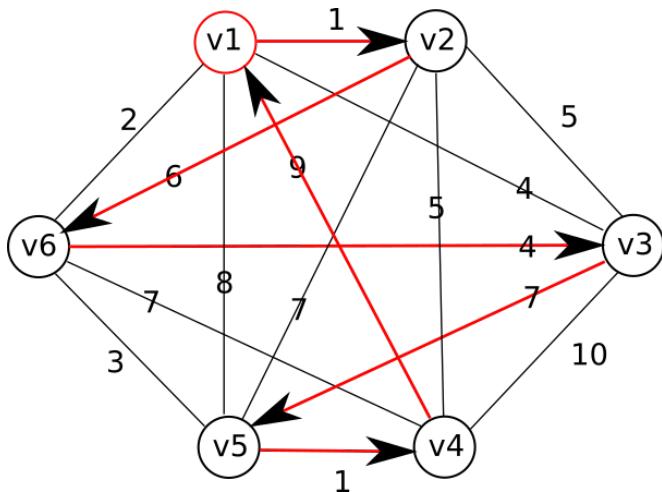
Passo 2: Tour de Euler em T

Note que embora todos os tours de Euler que possam ser extraídos pelo algoritmo de Fleury possuírem o mesmo peso, a ordem de visita dos vértices acaba por afetar o resultado final, uma vez que eles determinam quais “atalhos” iremos tomar.

$$L = \{v1, v2, v1, v6, v3, v6, v5, v4, v5, v6, v1\}$$

Passo 3: Eliminar repetições (equivale a tomar atalhos no grafo)

$$H = \{v1, v2, v6, v3, v5, v4, v1\}$$



$$\text{peso do ciclo é } w(H) = 1 + 6 + 4 + 7 + 1 + 9 = 28$$

Prop: A solução obtida pelo algoritmo Twice-Around em grafos Euclidianos satisfaz:

$$f(x_{TA}) \leq 2 f(x^*)$$

É fácil verificar que se x^* é a solução ótima, então se subtrairmos uma aresta e de x^* , obtemos uma árvore T, ou seja, $T = x^* - e$

Também sabemos que essa árvore T tem peso maior ou igual que a MST do grafo e por isso temos:

$$f(x^*) > w(T) \geq w(T_{MST})$$

o custo de x^* é maior que de T pois o ciclo tem uma aresta a mais

Então temos que:

$$f(x^*) > w(T_{MST})$$

o que implica em

$$2f(x^*) > 2w(T_{MST}) \quad (*) \text{ (é o peso do tour de Euler extraído no passo 2)}$$

Se G é Euclidiano (tomar atalhos é sempre mais vantajoso devido a desigualdade triangular)

$$f(x_{TA}) \leq 2w(T_{MST}) \quad (**)$$

E portanto, combinando (*) e (**)

$$2f(x^*) > 2w(T_{MST}) \geq f(x_{TA})$$

Algoritmo 2 -Optimal (2-Opt)

- Método iterativo

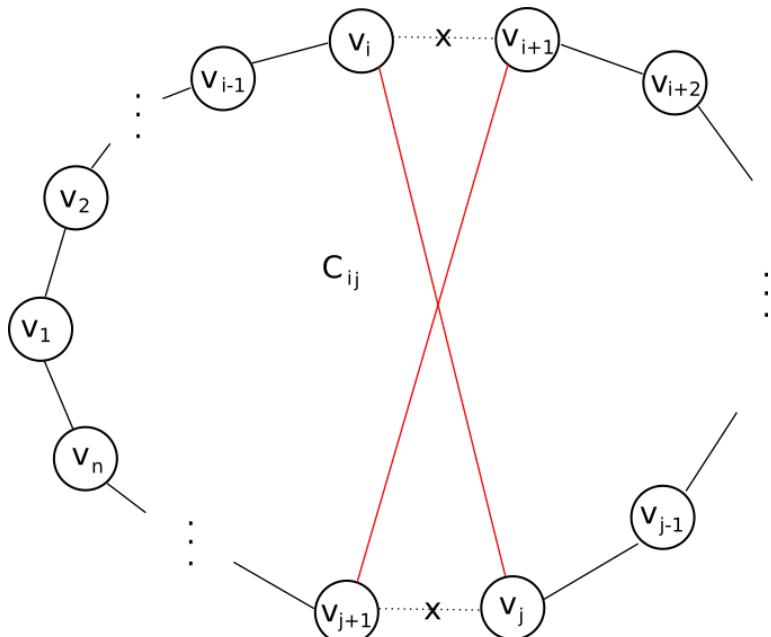
Ideia geral: partir de uma inicialização e tentar melhorá-la a cada passo (resultado final depende do chute inicial)

Obs: Não é força bruta! Há confusão pois os exemplos contém poucos vértices. Em casos com n grande a diferença é gritante.

Algoritmo

1. Seja $C = v_1v_2v_3 \dots v_nv_1$ uma solução inicial. Então $w(C) = w(v_1, v_2) + w(v_2, v_3) + \dots + w(v_n, v_1)$
2. Faça $i = 1$
3. Faça $j = i + 2$
4. Seja C_{ij} o ciclo

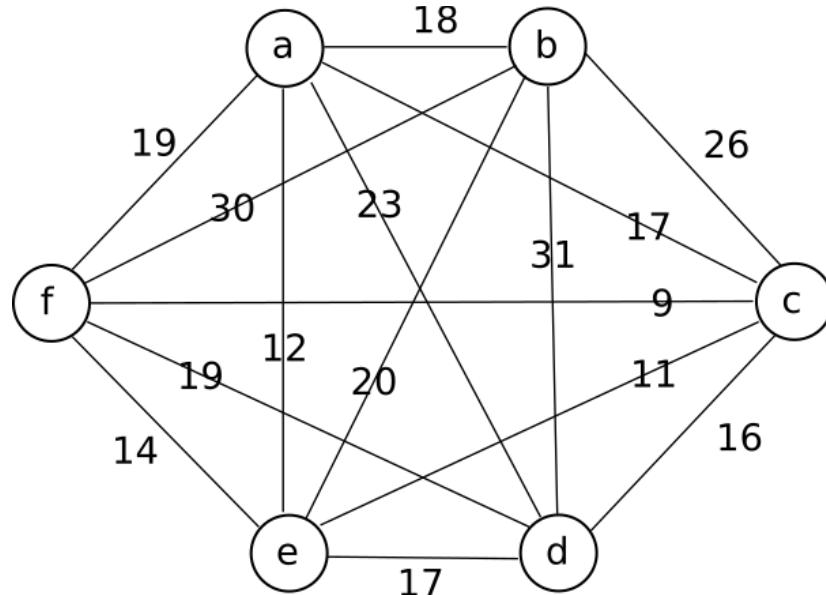
$$C_{ij} = v_1 v_2 \dots v_i v_j v_{j-1} v_{j-2} \dots v_{i+1} v_{j+1} \dots v_n v_1 \quad (\text{torção aplicada no ciclo } C)$$



Se $w(C_{ij}) < w(C)$ então $C = C_{ij}$ e volte para o passo 2. usando a nova sequencia. Caso contrário, siga para o passo 5.

5. Faça $j = j + 1$. Se $j \leq n$, volte para o passo 4. Caso contrário, $i = i + 1$ e se $i \leq n - 2$ volte para o passo 3, senão, PARE!

Ex:

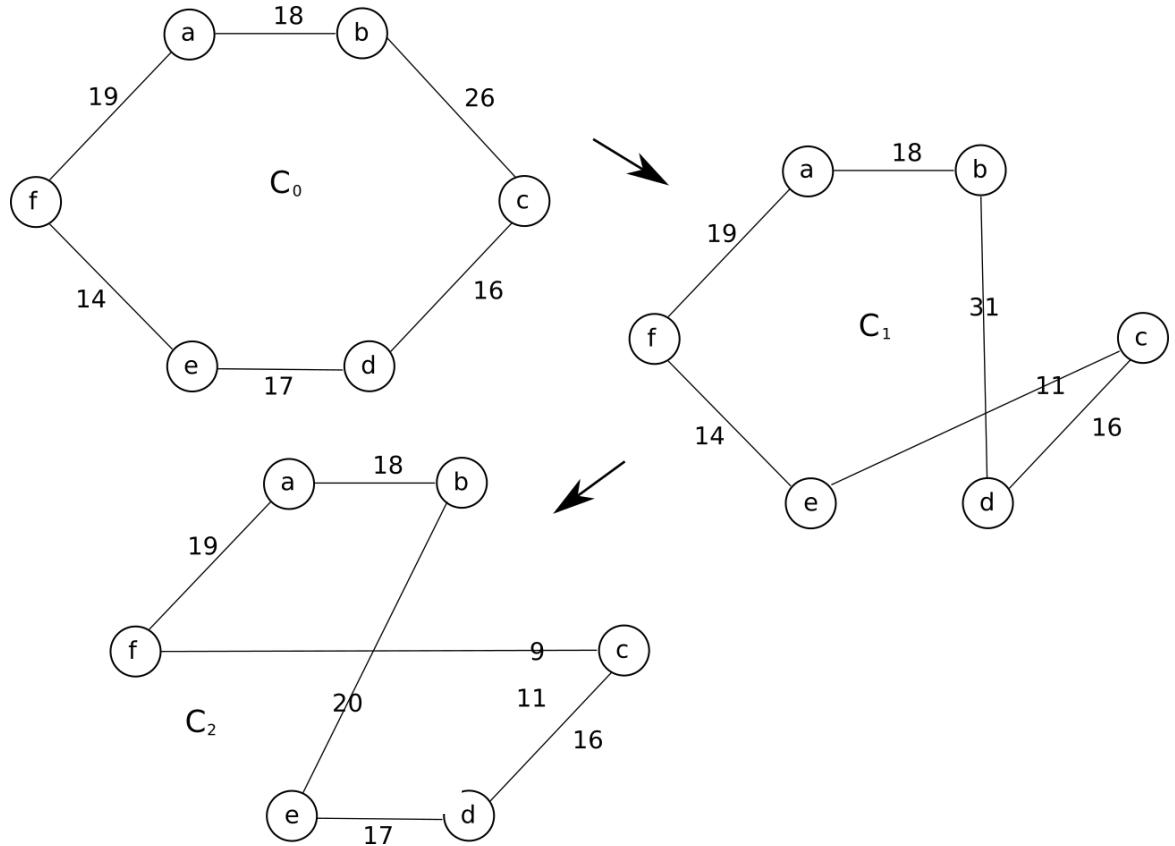


C								W	
i	j	1	2	3	4	5	6		
1	3	a	c	b	d	e	f	a	110
1	4	a	d	c	b	e	f	a	124
1	5	a	e	d	c	b	f	a	120
1	6	a	f	e	d	c	b	a	110
2	4	a	b	d	c	e	f	a	109 *
1	3	a	d	b	c	e	f	a	124
1	4	a	c	d	b	e	f	a	117
1	5	a	e	c	d	b	f	a	119
1	6	a	f	e	c	d	b	a	109
2	4	a	b	c	d	e	f	a	110
2	5	a	b	e	c	d	f	a	103 *
1	3	a	e	b	c	d	f	a	112
1	4	a	c	e	b	d	f	a	117
1	5	a	d	c	e	b	f	a	119
1	6	a	f	d	c	e	b	a	103
2	4	a	b	c	e	d	f	a	110

2	5	a	b	d	c	e	f	a	109
2	6	a	b	f	d	c	e	a	104
3	5	a	b	e	d	c	f	a	99 *
1	3	a	e	b	d	c	f	a	107
1	4	a	d	e	b	c	f	a	114
1	5	a	c	d	e	b	f	a	119
1	6	a	f	c	d	e	b	a	99
2	4	a	b	d	e	c	f	a	105
2	5	a	b	c	d	e	f	a	110
2	6	a	b	f	c	d	e	a	102
3	5	a	b	e	c	d	f	a	103
3	6	a	b	e	f	c	d	a	100
4	6	a	b	e	d	f	c	a	100

PARE!

Representação gráfica

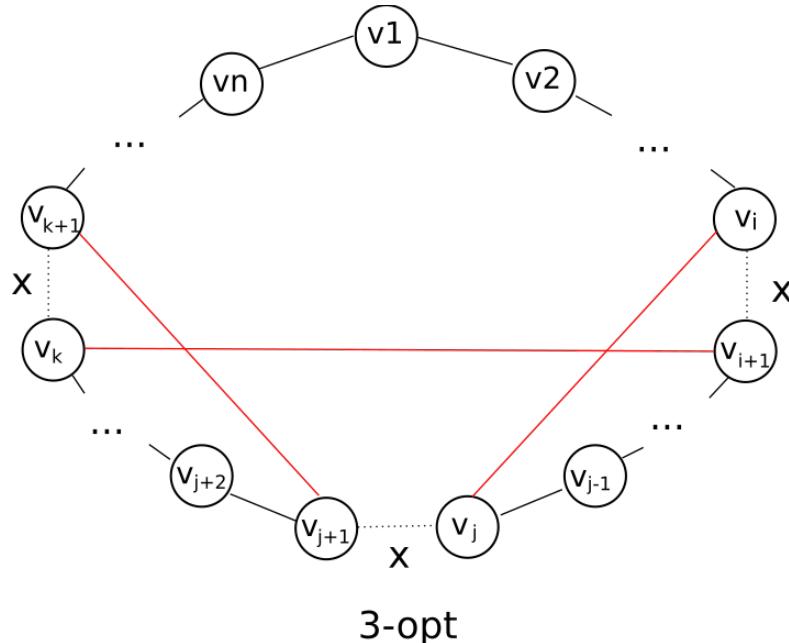


Prop: A solução obtida pelo algoritmo 2-Opt em grafos Euclidianos satisfaz:

$$f(x_{2-opt}) \leq \log n f(x^*)$$

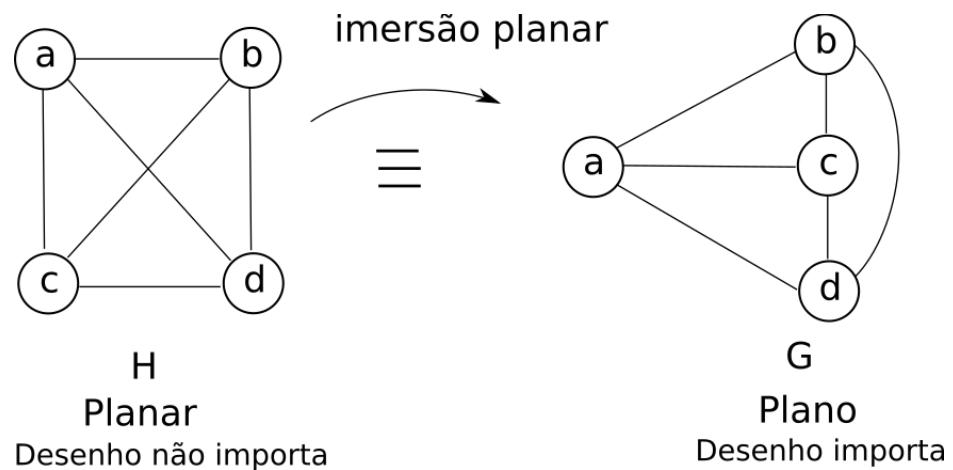
Obs: Para grafos genéricos (não necessariamente Euclidianos): $f(x_{2-opt}) \leq \sqrt{n} f(x^*)$

Obs: K-opt é a versão genérica do 2-opt, onde o número de cortes no ciclo é um número arbitrário K. Nesse caso, o número de combinações possíveis cresce, fazendo com que o custo computacional do algoritmo seja $O(n^K)$. Quanto maior K mais próximo da força bruta

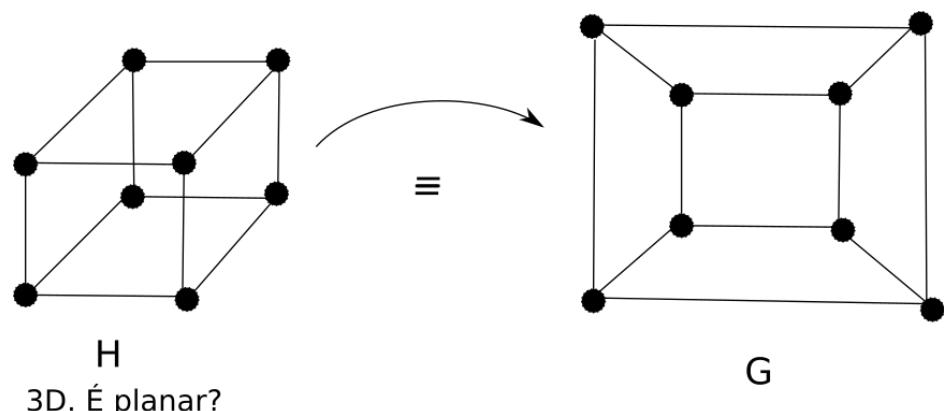


Grafos Planares

Def: Um grafo G é plano se ele pode ser desenhado numa superfície plana sem que haja cruzamento de arestas. G é planar se ele for isomorfo a um grafo plano.



Aspecto geométrico importa? Não (apenas topologia)



Importância:

1. Grafos planares são em geral esparsos porém conexos
2. Simplificação de vários problemas NP em grafos planares
3. Engenharias: projetos de interligação (água, gás, tubulações)
4. Projeto de circuitos impressos

O Problema do compartilhamento de recursos (Utility problem)

Suponha que existam 3 residências em um plano e cada uma precisa ser conectada as tubulações de 3 companhias: gás, água e eletricidade. Sem utilizar uma terceira dimensão, é possível realizar todas as nove conexões necessárias sem que as linhas se interceptem, ou seja, é possível alimentar as 3 casas com os 3 recursos? Imagine como se o mapa fosse um matriz enorme onde ficam os objetos

No plano ou esfera não, mas num toro SIM!

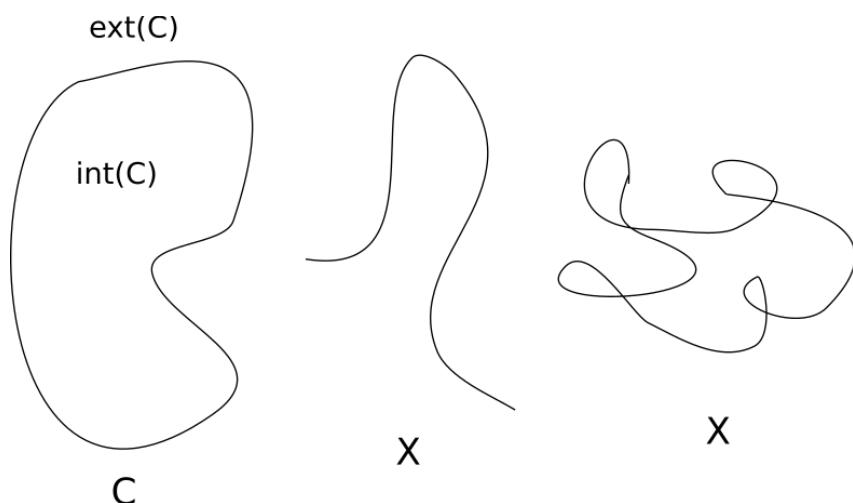
Habitantes da esfera e do toro, como distinguir a diferença (saber que há buraco no meio)

Problema: Dado um grafo G , como decidir se ele é planar ou não?

Queremos definir critérios que nos permitam responder a essa pergunta. O que eu devo procurar notar num grafo para saber se ele é planar ou não?

Def (Curva de Jordan):

Toda plana curva fechada que não intercepta a si própria



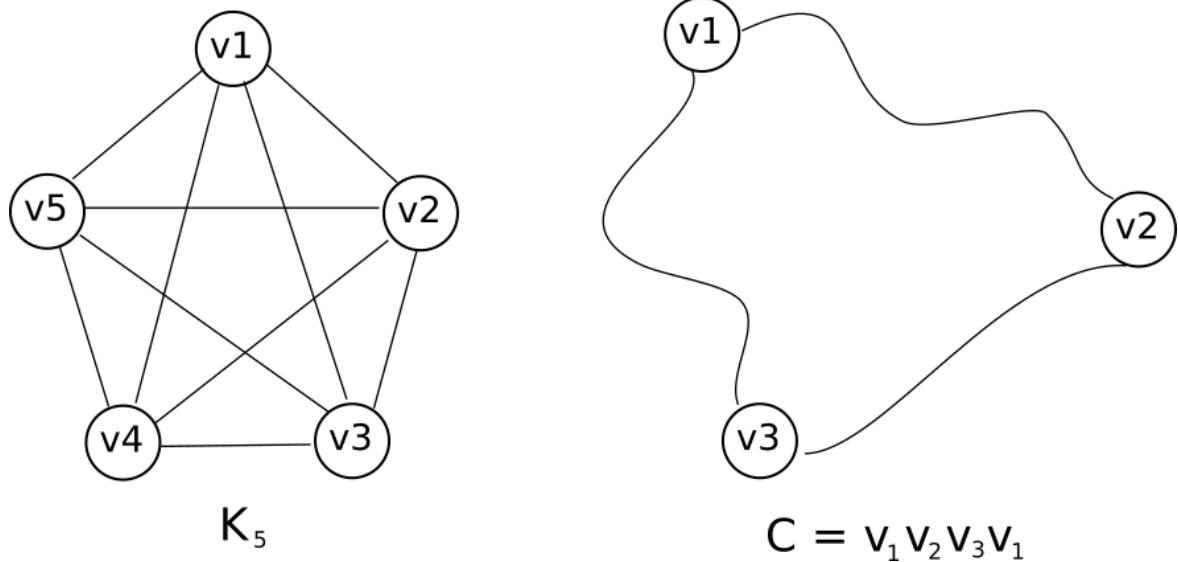
Obs: A noção de curva de Jordan para nós em grafos será a de ciclo. Todo ciclo pode ser visto como uma curva de Jordan uma vez que é uma trajetória fechada e que não repete vértices

Prop: Se C é uma curva de Jordan com $x \in \text{int}(C)$ e $y \in \text{ext}(C)$ então qualquer curva que une x a y intercepta C .

Queremos encontrar pistas sobre quais os menores blocos não planares que existem. Nessa busca, pode-se verificar o seguinte fato.

Teorema: O grafo completo K_5 não é planar

Iremos assumir que ele é planar. Veremos então que chegamos num absurdo, o que contradiz a hipótese inicial



Seja $C = v_1 v_2 v_3 v_1$ (curva de Jordan)

Como $v_4 \notin C$ temos 2 opções:

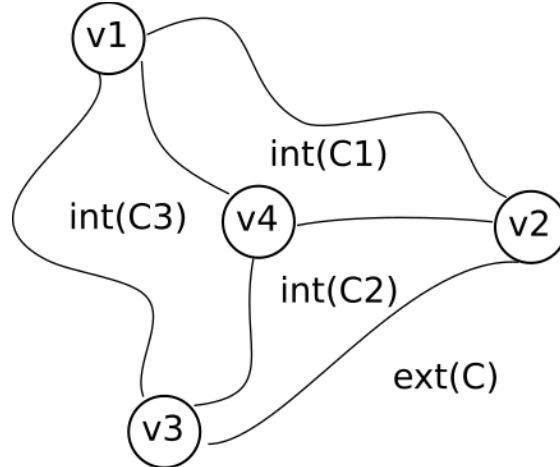
- a) $v_4 \in \text{int}(C)$
- b) $v_4 \in \text{ext}(C)$

Vamos primeiramente supor a), ou seja, $v_4 \in \text{int}(C)$. Assim, temos

$$C_1 = v_1 v_2 v_4 v_1$$

$$C_2 = v_2 v_3 v_4 v_2$$

$$C_3 = v_3 v_4 v_1 v_3$$

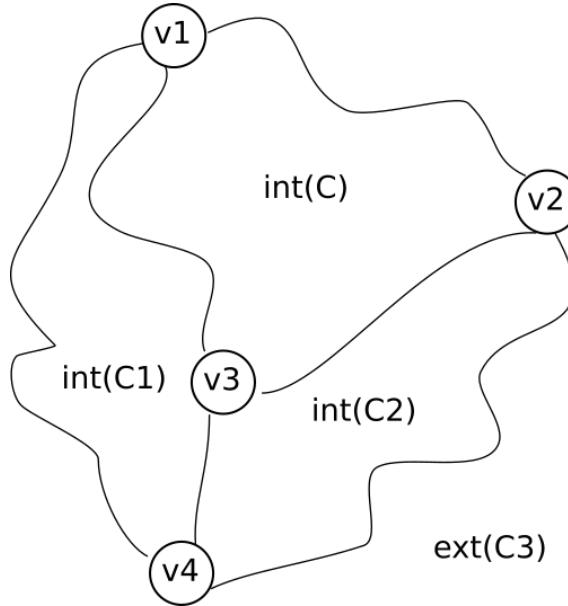


Então, temos a subdivisão do plano em 4 regiões. Para v_5 temos:

- i) $v_5 \in \text{ext}(C)$: aresta (v_4, v_5) cruza
- ii) $v_5 \in \text{int}(C_1)$: aresta (v_3, v_5) cruza
- iii) $v_5 \in \text{int}(C_2)$: aresta (v_1, v_5) cruza
- iv) $v_5 \in \text{int}(C_3)$: aresta (v_2, v_5) cruza

Senso assim, para v_4 no interior de C , não é possível K_5 ser planar. Vamos agora ao caso b), onde $v_4 \in \text{ext}(C)$. Assim, temos:

$$\begin{aligned} C &= v_1 v_2 v_3 v_1 \\ C_1 &= v_1 v_3 v_4 v_1 \\ C_2 &= v_2 v_3 v_4 v_2 \\ C_3 &= v_1 v_2 v_4 v_1 \end{aligned}$$

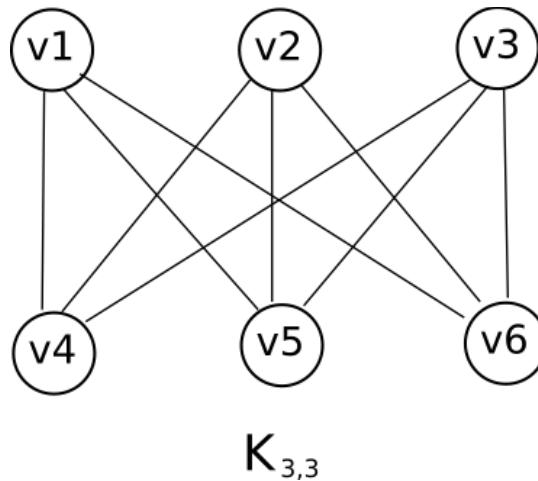


Novamente, temos a partição do plano em 4 regiões, de modo que para v_5 tem-se:

- i) $v_5 \in \text{int}(C)$: aresta (v_4, v_5) cruza
- ii) $v_5 \in \text{int}(C_1)$: aresta (v_2, v_5) cruza
- iii) $v_5 \in \text{int}(C_2)$: aresta (v_1, v_5) cruza
- iv) $v_5 \in \text{ext}(C_3)$: aresta (v_3, v_5) cruza

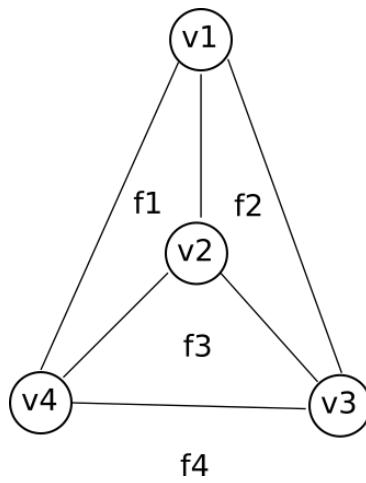
Portanto, não há como K_5 ser planar.

Prop: O grafo bipartido $K_{3,3}$ não é planar



Prop1: Fórmula de Euler

Seja G um grafo conexo e plano. Então $n - m + f = 2$, onde $n = |V|$, $m = |E|$ e f é o n. de faces



Prop2: Se G é planar com $|V| = n$ e $|E| = m$ então $m \leq 3(n - 2)$

Obs: Interessantes mas não podem ser usadas para decidir se G é planar. (para ser aplicado o grafo precisa estar desenhado como plano, então já preciso saber que é planar. Pouca utilidade prática.)

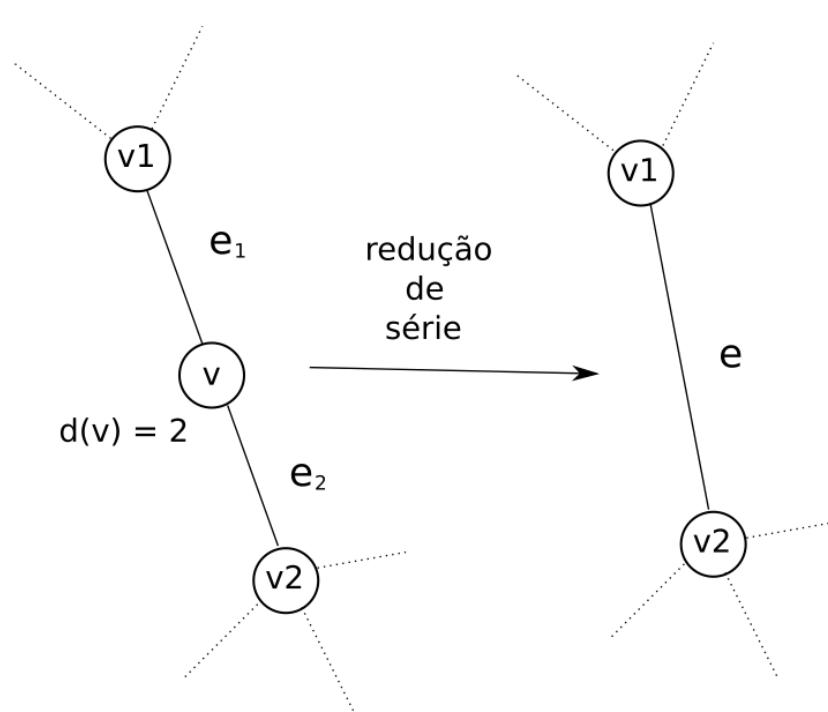
Teorema de Kuratowski (1930)

Descreve como determinar se G é planar ou não através de 3 operações básicas

- i) remoção de arestas
- ii) remoção de vértices
- iii) redução de séries

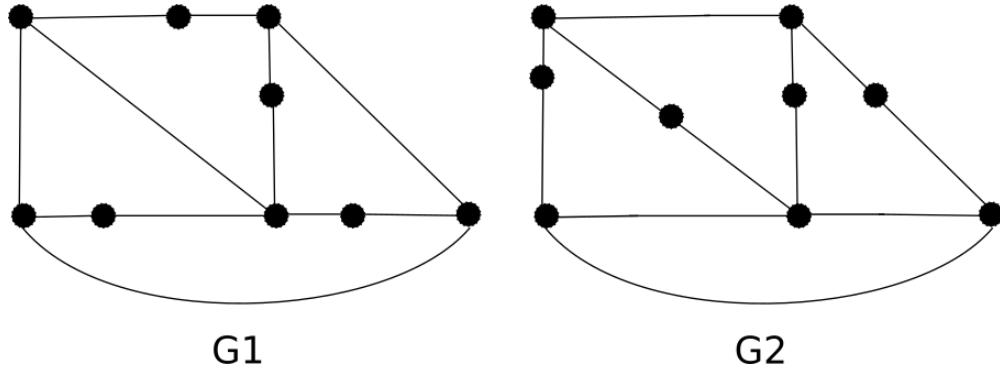
Def: Redução de série

Se um grafo G tem um vértice v de grau 2 e arestas (v_1, v) e (v, v_2) com $v_1 \neq v_2$ diz-se que as arestas (v_1, v) e (v, v_2) estão em série

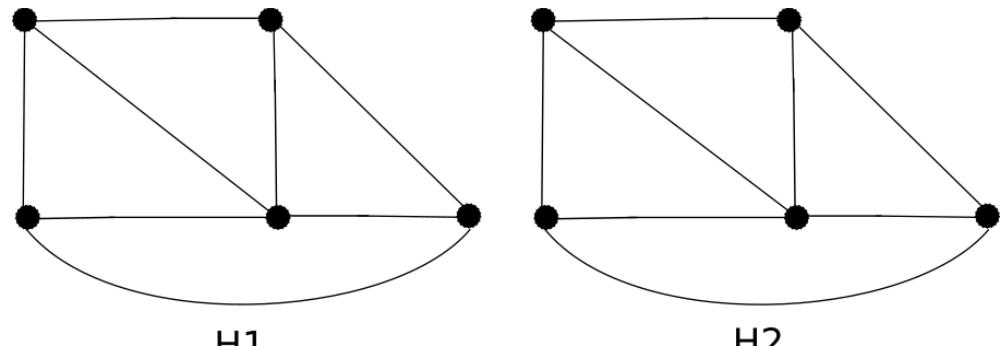


Def: G_1 e G_2 são homeomorfos se puderem ser reduzidos a grafos isomorfos a partir de reduções de série

Por exemplo, os dois grafos a seguir são homeomorfos.



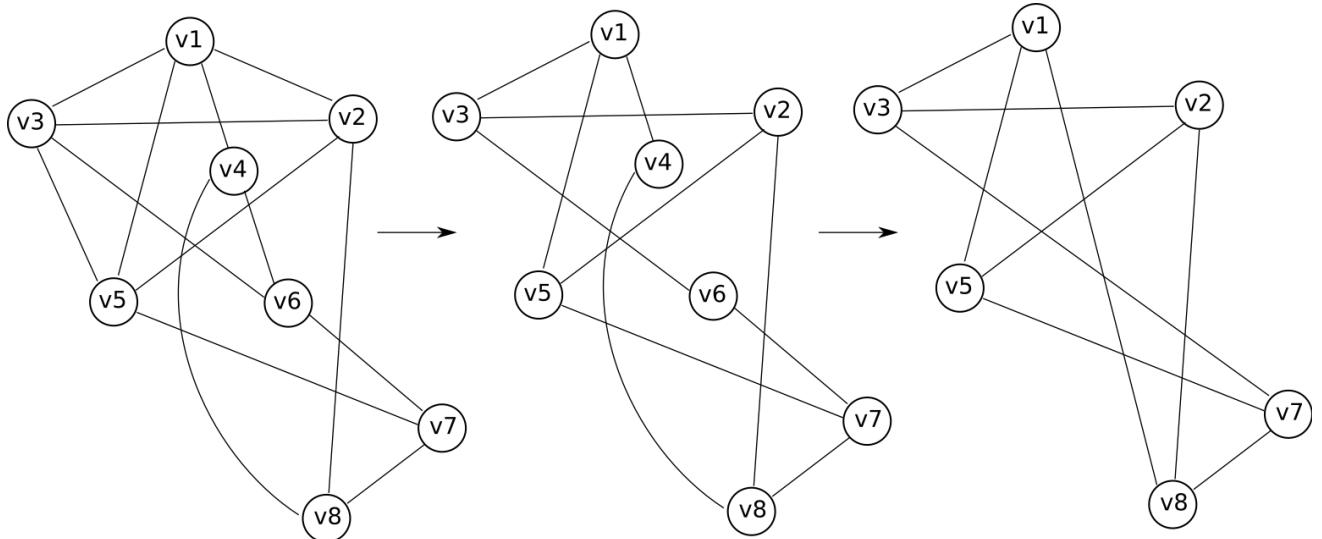
Isso pode ser facilmente percebido uma vez que ao aplicarmos reduções de séries em ambos, obtemos o seguinte resultado, ou seja, H1 é isomorfo a H2



Teorema
de Kuratowski

Um grafo G é planar se e somente se G não tiver um subgrafo homeomorfo a K_5 ou $K_{3,3}$

Critério objetivo, porém de difícil aplicação prática (algoritmos). (ele fala o que fazer mas não como fazer exatamente, os passos... cada caso é um caso)



1. Remoção de arestas: $(v1, v2)$, $(v3, v5)$ e $(v4, v6)$
2. Redução de séries: $v1 - v4 - v8$ e $v3 - v6 - v7$

O grafo resultante claramente é o $K_{3,3}$

Comentários:

Planarity testing - algoritmos para testar a planaridade de grafos: existem $O(n)$

- Path addition method (Hopcroft, Tarjan, 1974)

Planarity, o jogo

<http://planarity.net/>

Puzzles são resolvidos em $O(n)$ pelo computador mas para humanos leva-se muito mais tempo

Problema (Gênio indomável): Liste todas as árvores homeomorfas irredutíveis de 10 vértices

LINK: https://www.youtube.com/watch?v=iW_LkYiuTKE

Coloração de vértices

Área com diversas aplicações práticas (como veremos a seguir)

Def: Seja $G = (V, E)$ um grafo e P_1, P_2, \dots, P_n uma partição de V . Dizemos de P_i é um subconjunto independente se quaisquer 2 vértices de P_i não compartilham arestas (não são vizinhos).

Objetivo: Particionar o conjunto V no menor número possível de subconjuntos independentes

Def: Uma k -coloração de G atribui uma de K cores a cada vértice de G , de modo que a vértices adjacentes sempre são atribuídas cores/rótulos diferentes

P_1 : subconjunto dos vértices de cor 1

P_2 : subconjunto dos vértices de cor 2

...

P_k : subconjunto dos vértices de cor k

Pergunta: Dado um grafo $G = (V, E)$, \exists uma k -coloração para G (com $k > 2$) ? NP-Completo

Def: O número mínimo k para o qual existe uma coloração de G é o número cromático de G , denotado por $\chi(G)$ (é um atributo do grafo) .

Descobrir qual é o $\chi(G)$ dado um grafo G qualquer é NP-Hard

Propriedades: (nos ajudam a limitar os valores de $\chi(G)$ em problemas reais: limites inferiores e superiores)

a) Se $|V| = n$, então $\chi(G) \leq n$

i) Se G é o K_n , então $\chi(G) = n$

ii) Se G contém K_m como subgrafo, então $\chi(G) \geq m$

b) G é bipartido $\Leftrightarrow \chi(G) = 2$

Poscomp: quanto vale $\chi(K_{3791,7583})$?

c) Seja $G = (V, E)$ e $\Delta(G) = \max\{d(v) : v \in V\}$ (grau máximo). Então, $\chi(G) \leq \Delta(G) + 1$

d) (Teorema das 4 cores) Todo grafo G planar possui $\chi(G) \leq 4$

Primeiro resultado teórico que contou com a ajuda de computadores para enumerar um conjunto de possibilidades. Foi muito importante na cartografia e produção de mapas (primórdios da imprensa)

Algoritmos para coloração de vértices

Objetivo: Dado G , obter $\chi(G)$

Problema NP-completo: não há garantias de que sempre retornem o verdadeiro $\chi(G)$

Algoritmo Welsh & Powell (na maioria dos casos consegue obter $\chi(G)$ real)

1. Organizar os vértices de G em ordem decrescente de grau (ordem de acesso)
2. Para cada vértice v_i seja $C_i = \{1, 2, 3, \dots, i\}$ sua lista de cores (quanto antes o vértice, ou seja, maior o grau, menor sua lista de cores)
3. Faça $i = 1$
4. Seja c_i a 1ª cor em C_i . Atribua c_i a v_i
5. Para cada vizinho v_j de v_i ($j > i$)
 - a) Faça $C_j = C_j - \{c_i\}$
 - b) Faça $i = i + 1$ e se $i \leq n$ volte para 4

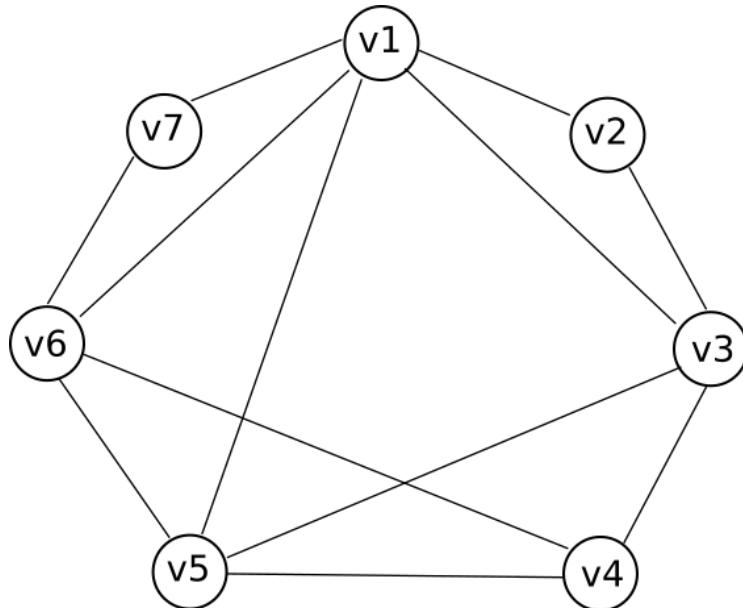
Ex: O Problema da Alocação de Frequências

Considere $N=7$ antenas de transmissão de sinais de telefonia. Sabe-se que devido a fatores como localização geográfica e tipo de serviço oferecido, as antenas possuem regiões de influência de modo que tem-se a seguinte matriz de interferências:

	v1	v2	v3	v4	v5	v6	v7
v1	x	x		x	x	x	
v2	x		x				
v3	x	x		x	x		
v4			x		x	x	
v5	x		x	x		x	
v6	x			x	x		
v7	x				x		

Essa matriz indica quando duas antenas interferem uma na outra com um x. Pergunta-se:

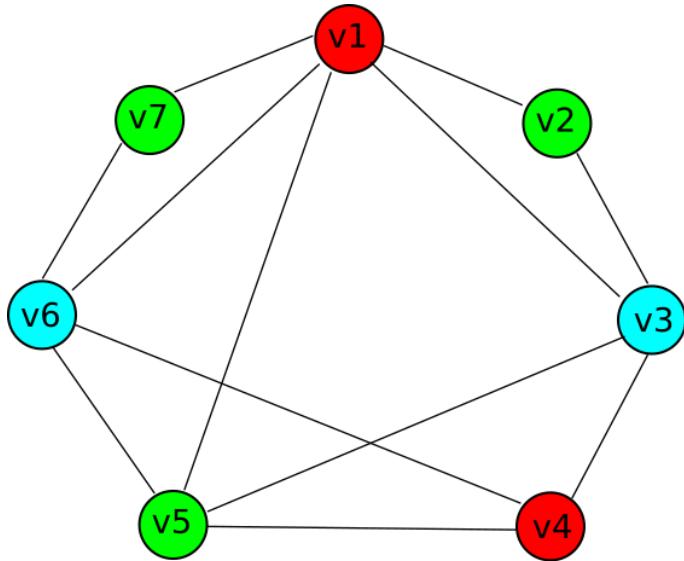
- i) Qual o menor número de frequências necessárias para que a comunicação se dê de forma correta?
- ii) As antenas devem operar em qual configuração de frequência?



Grafo de Incompatibilidade (ligar tudo que quero que fique separado)

Ordenação: v1, v3, v5, v6, v4, v2, v7

	C1={1}	C3={1,2}	C5={1,2,3}	C6={1,2,3,4}	C4={1,2,3,4,5}	C2={1,2,3,4,5}	C7={1,2,3,4,5}
v1 ← 1	x	C3={2}	C5={2,3}	C6={2,3,4}	C4={1,2,3,4,5}	C2={2,3,4,5}	C7={2,3,4,5}
v3 ← 2		x	C5={3}	C6={2,3,4}	C4={1,3,4,5}	C2={3,4,5}	C7={2,3,4,5}
v5 ← 3			x	C6={2,4}	C4={1,4,5}	C2={3,4,5}	C7={2,3,4,5}
v6 ← 2				x	C4={1,4,5}	C2={3,4,5}	C7={3,4,5}
v4 ← 1					x	C2={3,4,5}	C7={3,4,5}
v2 ← 3						x	C7={3,4,5}
v7 ← 3							x



$$X(G) = 3$$

$$\begin{aligned} P1 &= \{v1, v4\} \\ P2 &= \{v3, v6\} \\ P3 &= \{v2, v5, v7\} \end{aligned}$$

Outros problemas

Uma nova empresa aérea irá começar a operar com 7 aeronaves seguindo a programação de vôos (de A a G) definida pela tabela abaixo, sendo que todos os vôos partem de São Paulo e visitam cada uma das cidades listadas nas rotas na sequência em que elas aparecem:

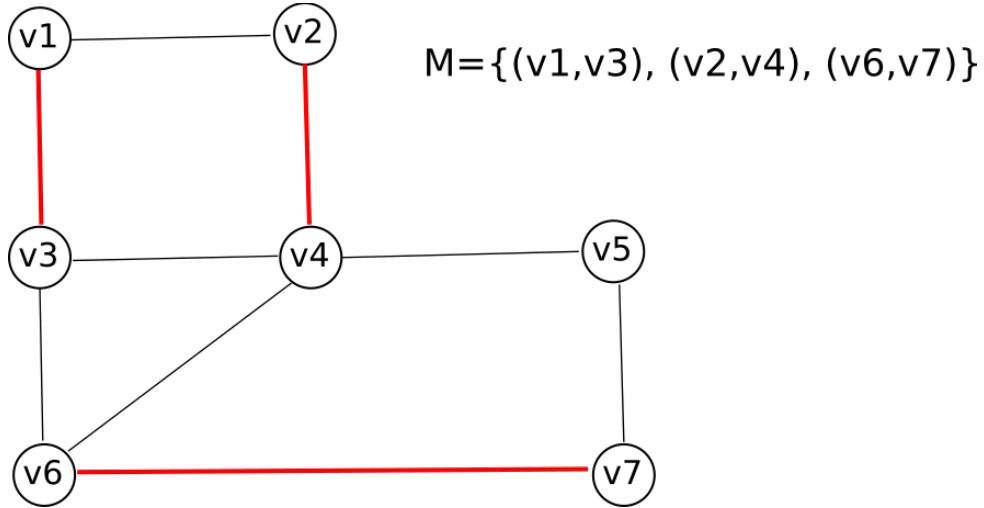
Vôo	Rota
A	Florianópolis – Rio de Janeiro – Natal – Fortaleza
B	Curitiba - Campinas – Ribeirão Preto – Fortaleza
C	Belo Horizonte – Natal – Fortaleza – Manaus
D	Belo Horizonte – São José do Rio Preto – Rio de Janeiro
E	Belo Horizonte – Recife – Natal
F	Brasília – Ribeirão Preto – Fortaleza
G	Brasília - Presidente Prudente – Campinas

Devido ao número limitado de aeronaves, o diretor da companhia não quer mais de um vôo por dia visitando uma determinada cidade, ou seja, se dois vôos passam pela cidade X eles devem obrigatoriamente não estar alocados para o mesmo dia. Sendo assim, modelando o problema com um grafo, e utilizando coloração de vértices, determine o número mínimo de dias necessários para que a empresa opere de acordo com a sua política de funcionamento.

Emparelhamentos (Matchings)

Def: Seja $G = (V, E)$ um grafo . Um emparelhamento $M \subseteq E$ é um subconjunto de arestas não adjacentes (não compartilham vértices)

Em outras palavras, um emparelhamento é um conjunto independente de arestas



Def: Vértice M -saturado

É todo $v \in V$ que é extremidade de alguma aresta de M

Ex: $v_1, v_2, v_3, v_4, v_6, v_7$

Def: Vértice M -não-saturado

É todo vértice $v \in V$ que não é extremidade de aresta de M

Ex: v_5

Def: Emparelhamento perfeito

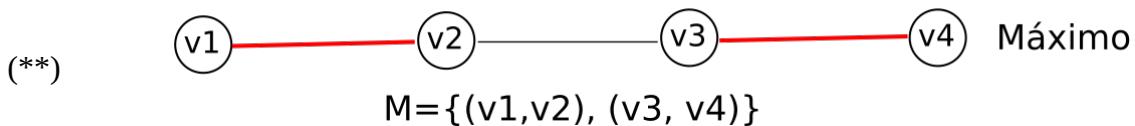
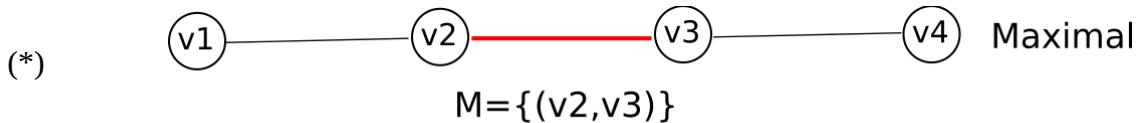
$\exists M \subset E$ tal que $\forall v \in V$ v é M -saturado $\rightarrow M$ é perfeito
(possível apenas se G tem um número par de vértices)

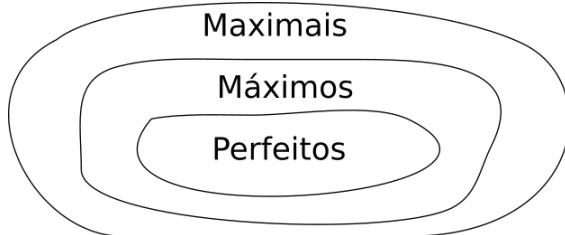
Def: Emparelhamento maximal

Se M não pode ser aumentado com o acréscimo de uma aresta então M é maximal

Def: Emparelhamento máximo

Se M é um emparelhamento de tamanho máximo dentre todos os possíveis, M é máximo





Def: Caminho M-alternado

Todo caminho P em que arestas estão alternadamente em M e E – M é um caminho M-alternado (**)

Def: Caminho M-aumentado

Todo caminho M-alternado em que tanto a origem quanto o destino são M-não -saturados (*)

Pergunta: Como podemos determinar se um emparelhamento M é máximo, ou seja, que ele o melhor possível para um dado grafo G? O resultado a seguir nos fornece a resposta.

Teorema de Berge (1957)

Um emparelhamento M em G é máximo \Leftrightarrow G não possui caminho M-aumentado

Prova: (ida)

M é máximo \rightarrow G não possui caminho M-aumentado =

G possui caminho M-aumentado \rightarrow M não é máximo

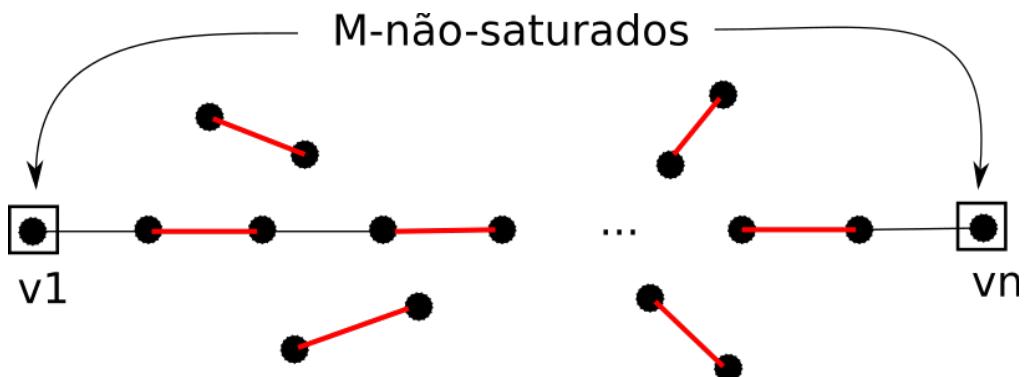
Seja um emparelhamento $M \subset E$. Então, há 2 tipos de arestas no grafo G:

- i) $\forall e \in M$: escura (faz parte de M)
- ii) $\forall e \notin M$: clara (não faz parte de M)

Suponha que P seja um caminho M-aumentado em G. Então P é da seguinte forma:

$P = \text{clara, escura, clara, escura, clara, escura, ..., escura, clara}$
(1^a) (última)

Pela alternância das arestas, temos que para cada clara devemos ter uma escura. Então se existem n pares de arestas (clara, escura) deve haver uma última aresta clara no final, de modo que o número total de arestas de P é da forma $2n + 1$. Note que é um número ímpar não importa o valor de n (há portanto uma aresta clara a mais que escura).



arestas de M que não estão em P
não podem incidir em vértices de P

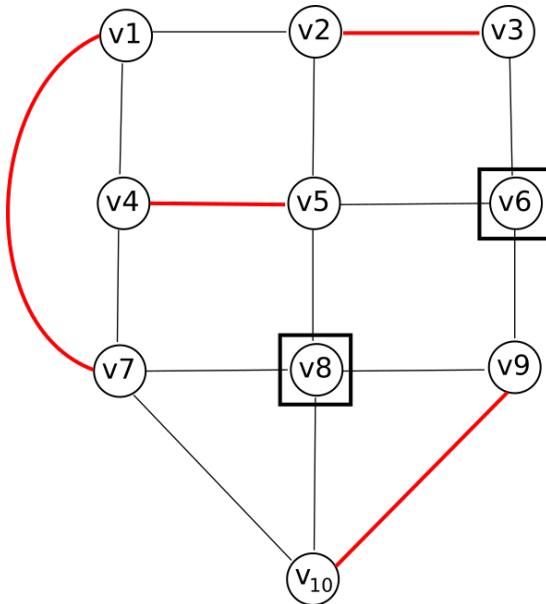
Assim, podemos definir um novo emparelhamento M' como:

$$M' = \{ \begin{array}{l} \forall e \in M \text{ que não está em } P \\ (\text{escura}) \end{array} \} \cup \{ \begin{array}{l} \forall e \in E - M \text{ que estão em } P \\ (\text{clara}) \end{array} \}$$

de modo que $|M'| = |M| + 1$

Obs: A operação que consiste em transformar M em M' usando P é chamada de “Transferência ao longo do caminho M -aumentado P ”

Ex:



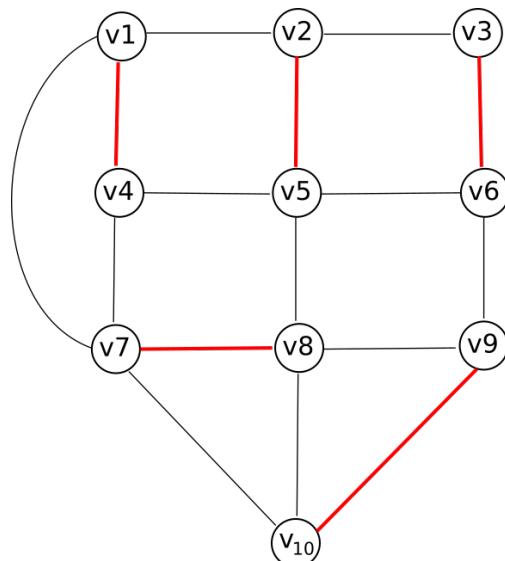
$$M = \{(v1, v7), (v2, v3), (v4, v5), (v9, v10)\}$$

M é máximo? Porque? Justifique

Note que existe o caminho M-aumentado $P = v6\ v3\ v2\ v5\ v4\ v1\ v7\ v8$, portanto M não é máximo. Aplicando a transferência ao longo do caminho M-aumentado P, iremos manter $(v9, v10)$ (pois não pertence ao caminho P, e inverter as arestas do caminho P, gerando:

$$M' = T(P) = T(v6 \ v3 \ v2 \ v5 \ v4 \ v1 \ v7 \ v8) = \{(v3, v6), (v2, v5), (v4, v1), (v7, v8), (v9, v10)\}$$

Note que o número de arestas em M' é uma unidade maior do que o número de arestas em M . E o novo emparelhamento M' , o que podemos dizer sobre ele?



Agora, sabemos como melhorar um emparelhamento M a partir de caminhos M -aumentados. Porém, ainda não vimos um método automatizado para buscar por caminhos M -aumentados em grafos. Esse é na verdade um problema bastante complexo para grafos arbitrários. Por motivos de simplificação, iremos adotar a hipótese de que estamos lidando com grafos bipartidos. Isso, apesar de uma limitação, não representa um problema, dado que a grande maioria dos problemas práticos envolvendo emparelhamentos são definidos em grafos bipartidos (problema do casamento, alocação, atribuição de recursos, ...)

Diante do exposto, precisamos saber sob que condições um grafo bipartido admite um emparelhamento máximo, ou seja, um *matching* que sature todos os vértices da partição escolhida, por exemplo X . Para isso, iremos apresentar o teorema do casamento, que nos fornece um critério objetivo para a existência de emparelhamentos máximos em grafos bipartidos.

O Teorema do Casamento (Hall, 1935)

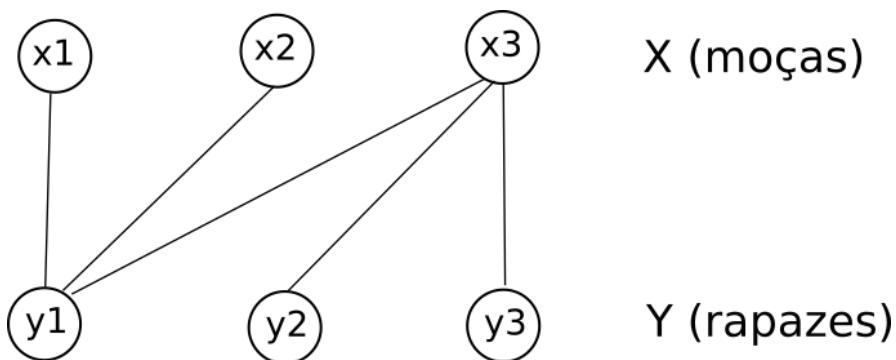
Seja G um grafo bipartido com $V = X \cup Y$, $X \cap Y = \emptyset$. G contém um emparelhamento M que satura $\forall v \in X$, se e somente se, para todo subconjunto S de X tivermos:

$|N(S)| \geq |S|$ onde $N(S)$ denota o conjunto vizinhança de S (todos elementos alcançáveis a partir dos elementos de S)

Ou seja, a vizinhança do conjunto S tem pelo menos tantos elementos quantos S tem

Em palavras, se um conjunto de n garotas conhece um conjunto de n rapazes a pergunta é: quando é possível que todas se casem. A condição nos diz que é possível que todas se casem se todo subconjunto de k garotas conhece coletivamente pelo menos k rapazes.

Ex:



O número de subconjuntos de um conjunto X é o número de elementos do conjunto potência denotado por $2^{|X|}$, que nesse caso é igual a 8. Esse é um dos problemas com esse resultado: enumerar todos os possíveis subconjuntos é computacionalmente inviável para n grande (complexidade $O(2^n)$ - exponencial)

$$\begin{aligned} S1 &= \{x1\}, N(S1) = \{y1\} \\ S2 &= \{x2\}, N(S2) = \{y1\} \\ S3 &= \{x3\}, N(S3) = \{y1, y2, y3\} \\ S4 &= \{x1, x2\}, N(S4) = \{y1\} \text{ (Falhou)} \end{aligned}$$

Portanto, nesse grafo não adianta tentar buscar emparelhamento M que sature todos os vértices de X (não existe tal M)

Dado um grafo G bipartido, sabemos decidir se existe um emparelhamento máximo M ou não. Agora, iremos responder a pergunta: como buscar caminhos M -aumentados?

Árvore M -alternada

- Estrutura utilizada para buscar caminhos M-aumentados. Uma árvore T é M-alternada se satisfaz:

- i) a raiz x_0 é M-não-saturada
- ii) $\forall v \in T$ o único caminho de x_0 a v é M-alternado

Veremos a seguir um método que combina o teorema de Berge, o teorema do casamento e árvores M-alternadas para a obtenção de emparelhamentos máximos em grafos bipartidos em tempo polinomial: trata-se do algoritmo Húngaro

O Método Húngaro

O padrão de crescimento de uma árvore M-alternada com raiz x_0 é tal que em qualquer estágio temos uma árvore de um de 2 possíveis tipos:

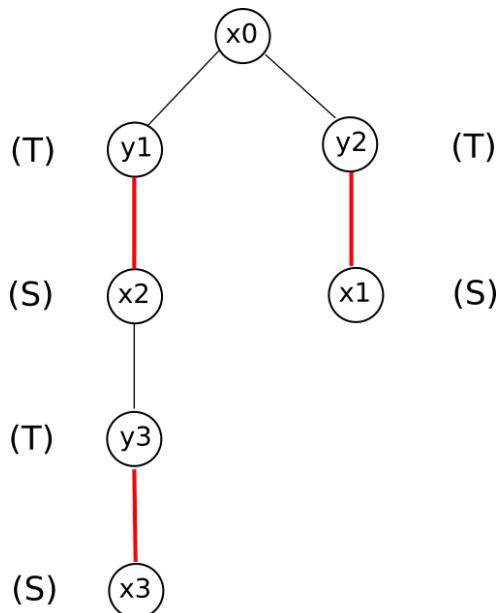
- i) Árvore do tipo-I: todos os vértices de T são M-saturados (com exceção da raiz x_0)
- ii) Árvore do tipo-II: T possui um vértice folha M-não-saturado

Veremos a seguir como as árvores do tipo-I estão relacionadas com o teorema do casamento

Análise das árvores do tipo-I

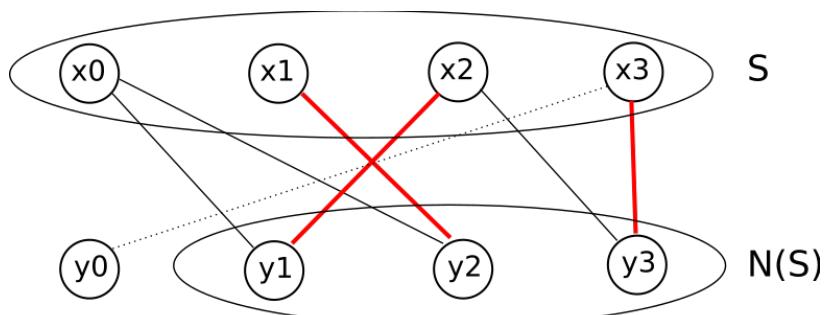
Sejam

- a) S' : conjunto dos vértices a uma distância par de x_0 (raiz)
- b) T : conjunto dos vértices a uma distância ímpar de x_0 (raiz)



Então, $|S'| = |T|$ (pois se todos os vértices são saturados, para cada elemento de Y tem que haver um de X correspondente – outro lado da aresta do emparelhamento). (Todo vértice a uma distância par de x_0 coloco no S e todo vértice a distância ímpar de x_0 coloco no T)

Define-se $S = S' \cup \{x_0\}$. Assim, $T \subseteq N(S)$ (olhando em G). Isso significa que pode ou não haver mais arestas para “pendurar” na árvore T.



Desse modo podemos dividir as árvores do tipo-I em 2 subcasos:

- a) $T = N(S)$ (não tem mais o que adicionar a árvore, impossível crescer T)
- b) $T \subset N(S)$ (posso continuar crescendo a árvore pois há arestas para adicionar)

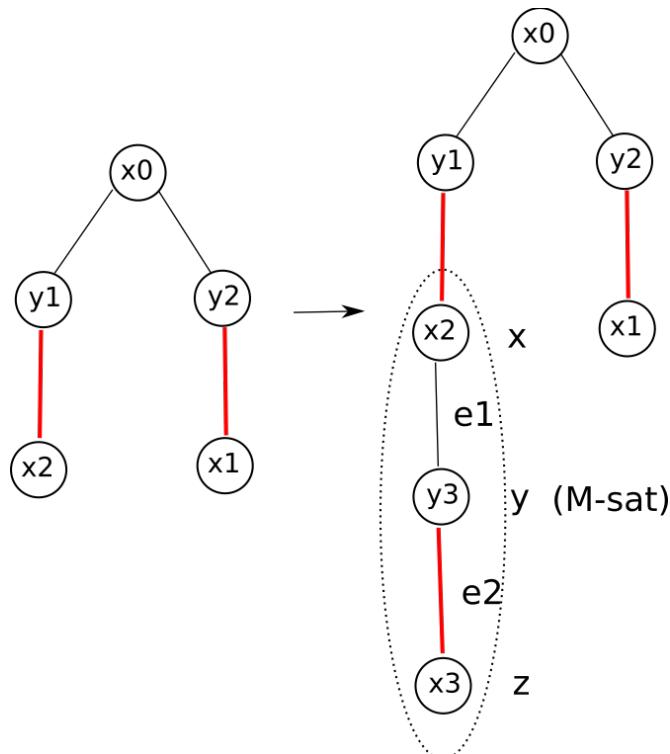
Subcaso a)

Se $T = N(S)$, então $|N(S)| = |T| = |S'| = |S| - 1$ (menos 1 por causa da raiz). Portanto, temos que $|N(S)| < |S|$, o que fere o teorema do casamento, pois para que exista um emparelhamento M que sature $\forall v \in X$, temos que ter $|N(S)| \geq |S|$. Em outras palavras, não há como melhorar o emparelhamento M atual. Devemos parar.

Subcaso b)

$$T \subset N(S) \Rightarrow |N(S)| > |T| \Rightarrow |N(S)| > |S'| \Rightarrow |N(S)| > |S| - 1 \Rightarrow |N(S)| \geq |S| \quad (\text{T. C.})$$

É possível melhorar emparelhamento M buscando caminho M-aumentado (há o que adicionar na árvore T). Nesse caso, $\exists y \in G$ que não está na árvore e é adjacente a algum $x \in S$. Porém, y pode ser de 2 tipos: M-saturado ou M-não-saturado. Se y é M-saturado, então \exists aresta $yz \in M$. Assim, crescemos a árvore adicionando 2 arestas: (x, y) e (y, z) , gerando uma nova árvore do tipo-I



Caso contrário, se y é M-não-saturado, então encontramos um caminho M-aumentado P da raiz x_0 até y ($M' = T(P)$) - árvore do tipo-II

Em resumo, a ideia do algoritmo Húngaro consiste em observar repetidamente as seguintes condições:

- | | |
|--------|---|
| Tipo-I | $T = N(S) \rightarrow \nexists M$ que satura $\forall v \in X$ (fere T. C.) |
| | $T \subset N(S) \rightarrow$ continue buscando caminho M-aumentado P |
-
- | | |
|---------|--|
| Tipo-II | Encontramos caminho M-aumentado $P: M' = T(P)$ |
|---------|--|

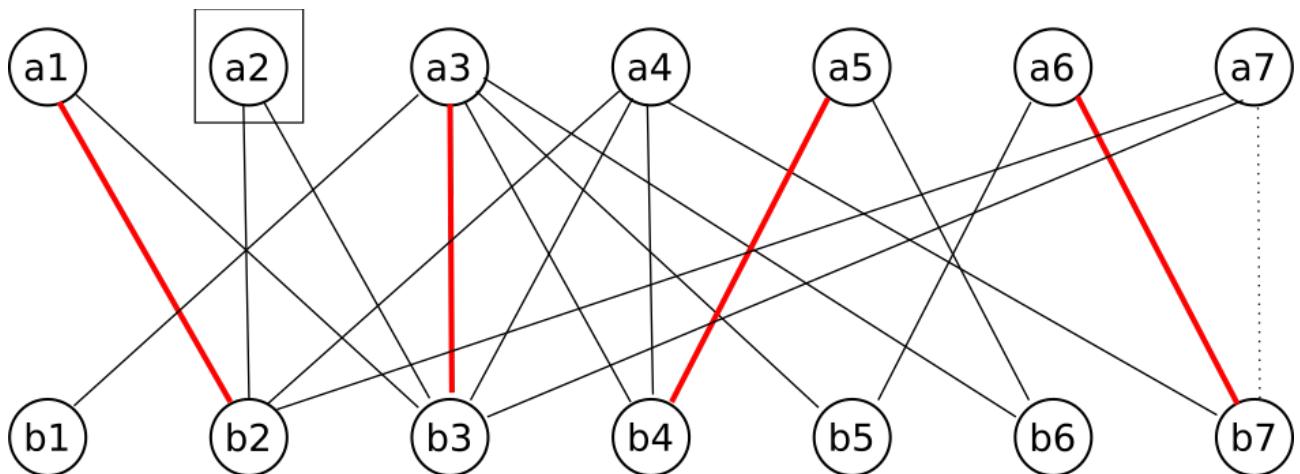
Algoritmo Húngaro

Entrada: $G = (V, E)$ bipartido + M inicial

Saída: M que satura $\forall v \in X$ (sucesso) ou S que fere T. C. (fracasso)

1. Se M satura $\forall v \in X$, pare e retorne M
Caso contrário, seja x_0 o 1º vértice M -não-saturado de X ainda não escolhido
Faça $S = \{x_0\}$ e $T = \emptyset$
2. Se $N(S) = T$, então sabemos que $|N(S)| < |S|$. Pare, pois S fere T. C. Retorne S
Caso contrário, escolha y como o 1º vértice da lista $N(S)$ tal que $y \notin T$ (não pertence a T)
3. Se y é M -saturado, seja $yz \in M$ a aresta que emparelha y a z .
Faça $S = S \cup \{z\}$, $T = T \cup \{y\}$ e volte para 2.
Se y é M -não-saturado, temos uma árvore do tipo-II e P de x_0 a y é um caminho M -aumentado. Faça $M' = T(P)$ (transf. ao longo do caminho P) e retorne para 1.

Ex: Um novo projeto a ser desenvolvido na empresa *Boogle* consiste num conjunto de 7 tarefas ($a_1, a_2, a_3, a_4, a_5, a_6, a_7$). A empresa possui na equipe de desenvolvimento apenas 7 profissionais ($b_1, b_2, b_3, b_4, b_5, b_6, b_7$). A partir de uma mapa de competências o gerente do projeto elaborou um modelo baseado em grafo para visualizar quais profissionais estão aptos a realizar quais tarefas. Sabendo que um profissional deve se associar a uma única tarefa, é possível alocar tarefas a pessoas de modo a completar todas as tarefas simultaneamente? Em caso positivo, forneça a alocação resultante. Em caso negativo, explique porque não é possível. (considere o *matching* inicial dado a seguir). Resolva o problema usando o algoritmo Húngaro.



$$M^{(0)} = \{a_1b_2, a_3b_3, a_5b_4, a_6b_7\}$$

i	S	T	$N(S)$	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
1	[a2]	\emptyset	[b2, b3]	b2 (M-sat)	(b2, a1)
	[a1, a2]	[b2]	[b2, b3]	b3 (M-sat)	(b3, a3)
	[a1, a2, a3]	[b2, b3]	[b1, b2, b3, b4, b5, b6]	b1	---

b1 (quem fez com que ele surgiu?) a3 b3 (quem fez com que ele surgiu?) a2

$$P' = b1 \ a3 \ b3 \ a2 \rightarrow P = \text{inv}(P') = a2 \ b3 \ a3 \ b1$$

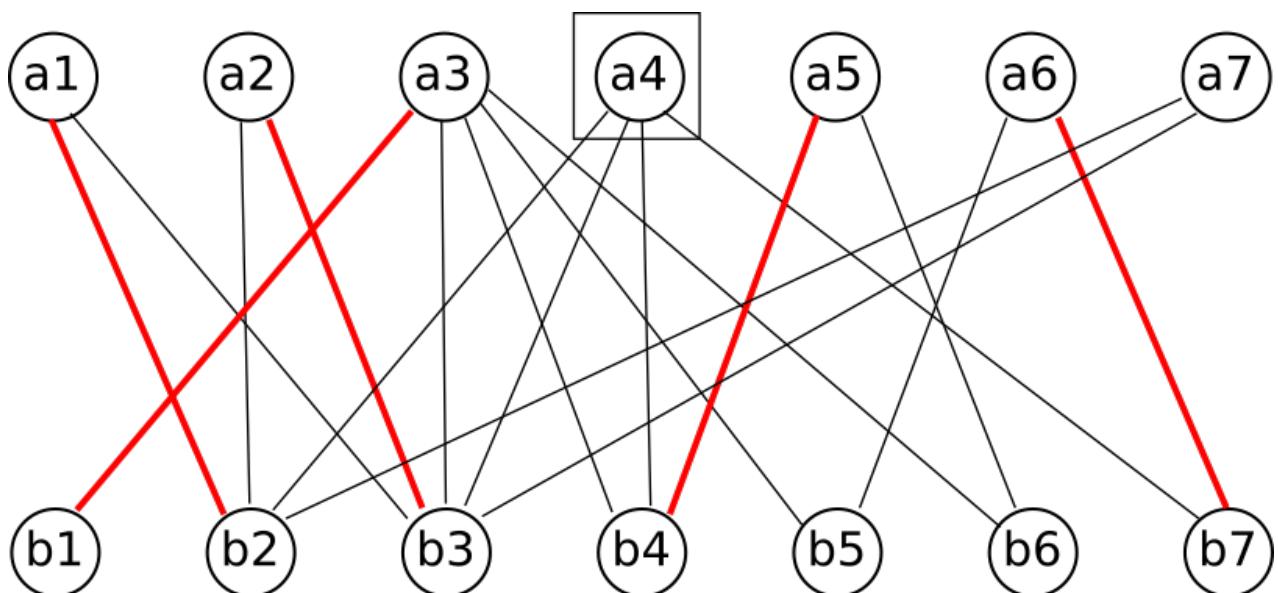
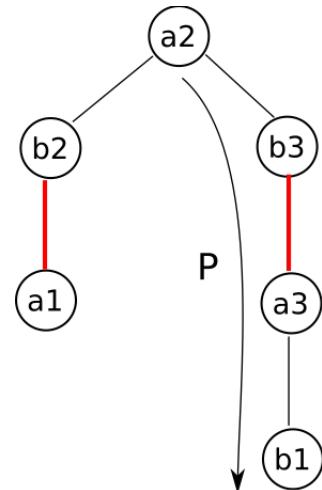
$M^{(1)} = T(P)$ (percorrer P ligando as arestas que não estão em $M^{(0)}$ e desligando as que estão)

a2 b3 = OK

b3 a3 = x

a3 b1 = OK

$$M^{(1)} = \{a1b2, a2b3, a3b1, a5b4, a6b7\}$$



i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
2	[a4]	\emptyset	[b2, b3, b4, b7]	b2 (M-sat)	(b2, a1)
	[a1, a4]	[b2]	[b2, b3, b4, b7]	b3 (M-sat)	(b3, a2)
	[a1, a2, a4]	[b2, b3]	[b2, b3, b4, b7]	b4 (M-sat)	(b4, a5)
	[a1, a2, a4, a5]	[b2, b3, b4]	[b2, b3, b4, b6, b7]	b6	---

$$P' = b6 \ a5 \ b4 \ a4 \rightarrow P = \text{inv}(P') = a4 \ b4 \ a5 \ b6$$

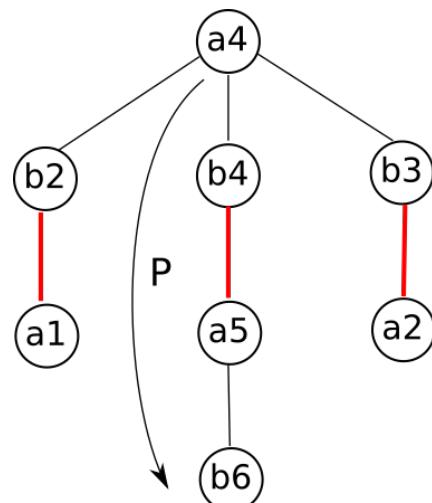
$M^{(2)} = T(P)$ (percorrer P ligando as arestas que não estão em $M^{(1)}$ e desligando as que estão)

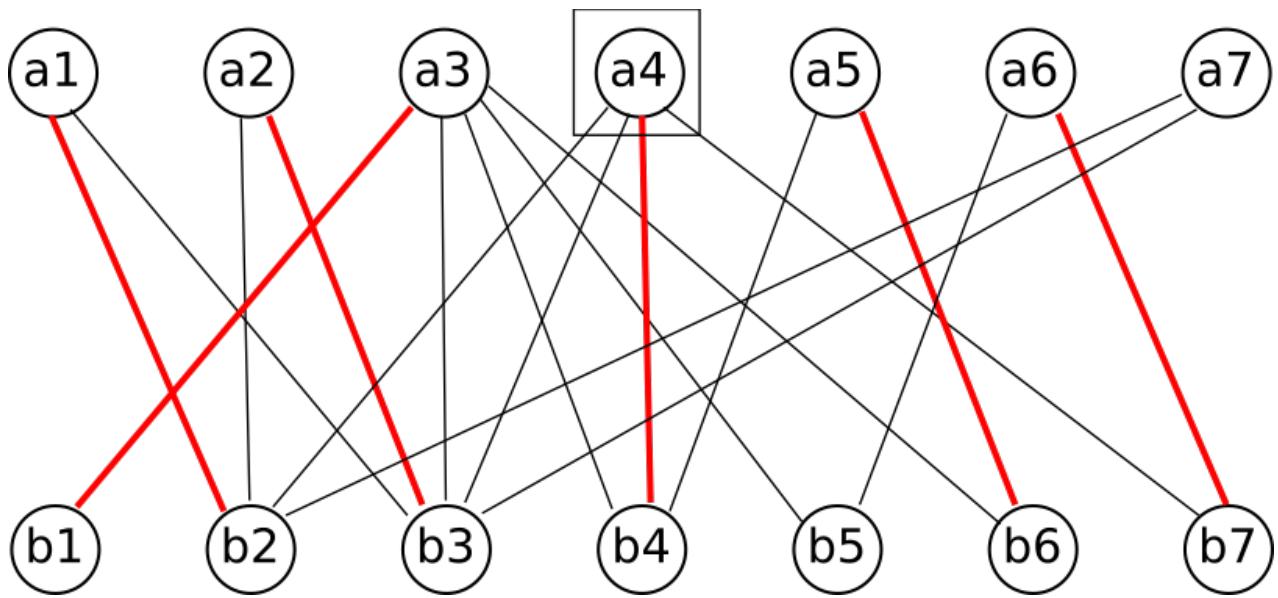
a4 b4 = OK

b4 a5 = x

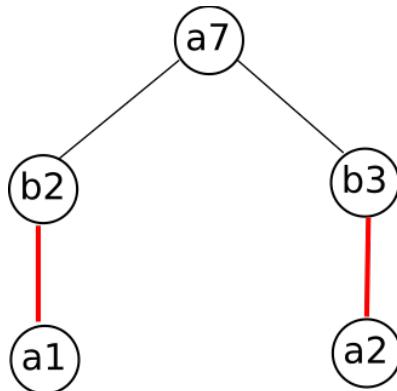
a5 b6 = OK

$$M^{(2)} = \{a1b2, a2b3, a3b1, a4b4, a5b6, a6b7\}$$

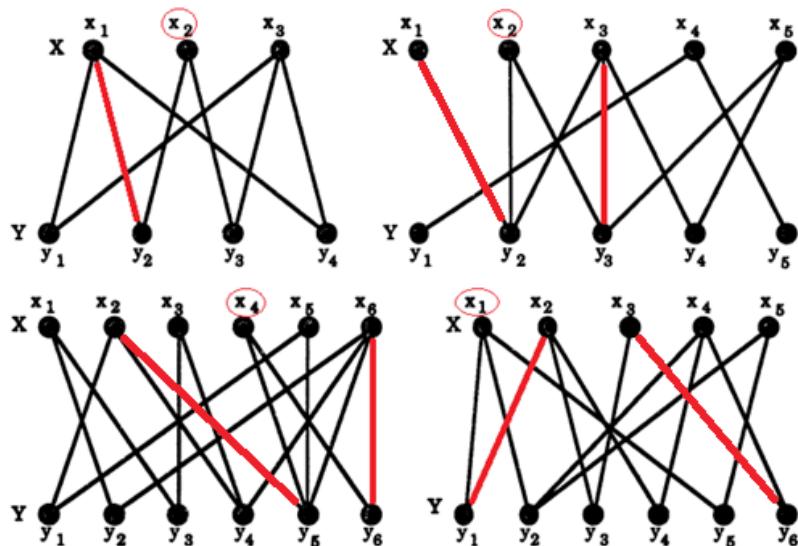




i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
3	[a7]	\emptyset	[b2, b3]	b2 (M-sat)	(b2, a1)
	[a1, a7]	[b2]	[b2, b3]	b3 (M-sat)	(b3, a2)
	[a1, a2, a7]	[b2, b3]	[b2, b3]	---	---



Como $N(S) = T$, temos que $S = \{a1, a2, a3\}$ fere o T. C. pois $N(S) = \{b2, b3\}$. Portanto, não há emparelhamento M que sature todo vértice de X. (se houvesse aresta a7b7, OK)



Emparelhamentos Estáveis

Área de interface entre Teoria dos Grafos e Teoria dos Jogos.

Objetivo: Dado um grafo bipartido completo, e um conjunto de listas de preferências/rankings, encontrar dentre todos os emparelhamentos possíveis, o mais estável (noção relacionada com a ideia de evitar futuros rompimentos)

Aplicações reais em diversas áreas:

- Sistemas de recomendação (candidatos/vagas)
- Alocação de alunos a universidades
- Alocação de residentes a hospitais
- Sistema para doação de órgãos

Prêmio Nobel em 2012 na área de ciências econômicas (Lloyd Shapley) por contribuições e profundo impacto no estudo, caracterização e regulamentação de sistemas econômicos (mercados) não controlados pelo capital. (a ideia básica é que nas situações em que o algoritmo Gale-Shapley se aplica não se obtém privilégios e vantagens impostas por escolher antes dos demais)

O problema do emparelhamento estável

Entrada:

$G = (V, E)$ bipartido completo $|X| = |Y| = n$ + 2n listas de preferência/rankings

Saída:

Emparelhamento perfeito S (quando $|X| = |Y|$, sempre é perfeito)

O conceito de estabilidade

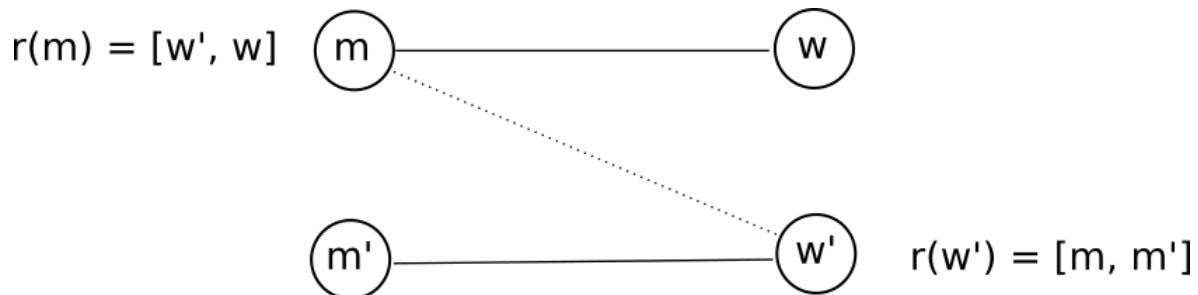
- Rankings / Listas de preferências: cada objeto de um lado deve ter uma lista em que rankeia todos os elementos do outro lado

$$\begin{aligned} \forall m \in M \quad & \exists r(m) = [w_1, w_2, \dots, w_n] \\ \forall w \in W \quad & \exists r(w) = [m_1, m_2, \dots, m_n] \end{aligned}$$

Definiremos um predicado ternário $P(m, w, w')$ para denotar que m prefere w a w' , ou seja, w vem antes de w' na lista $r(m)$

Def (Estabilidade):

Seja S o emparelhamento a seguir:



$$S = \{(m, w), (m', w')\}$$

S é estável?

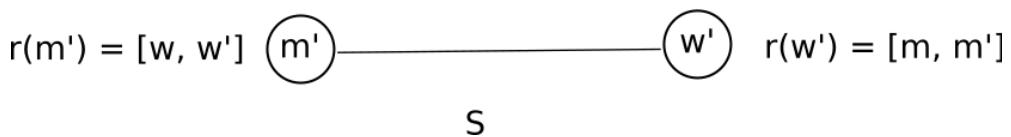
$\exists(m, w) \in S \wedge \exists(m', w') \in S$ tal que $P(m, w', w) \wedge P(w', m, m')$, portanto o par (m, w') define uma instabilidade em S .
Portanto, S não é estável.

Nada impede que m ou w' rompa o relacionamento atual de maneira unilateral e melhora sua condição, ou seja, na impede que se dê uma ruptura e após isso o outro se une ao parceiro predileto.

Obs: Note que $(m, w') \notin S$ (uma instabilidade nunca é um par que existe, mas sim um par que nada impede que ele possa aparecer no futuro)

Def: S é estável se S é perfeito e $\nexists(m, w)$ que provoque instabilidade

Ex:

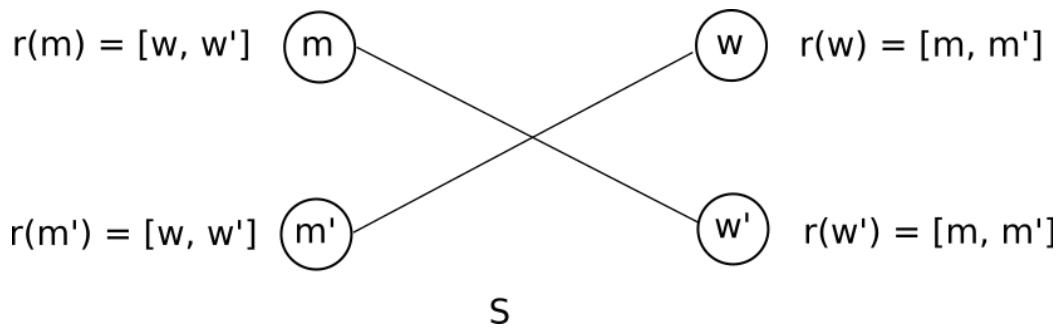


S é estável?

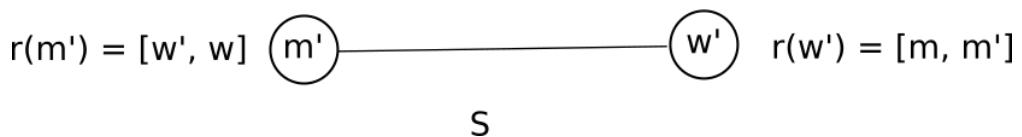
Sabemos que $P(m, w, w')$ e $P(w, m, m')$ e como o par (m, w) existe em S , eles não podem tentar nada melhor. Suponha que m' rompa unilateralmente com w' . Nesse caso, w vendo que m' está livre não irá largar de m para ficar com ele, pois m é o primeiro da lista. Da mesma forma, suponha que w' rompa unilateralmente com m' . O homem m não vai romper com w para ficar com w' pois prefere a parceira atual. Dessa forma, não há instabilidade que possa surgir e portanto S é estável.

Obs: Estabilidade não é agradar a todos com primeira opção.

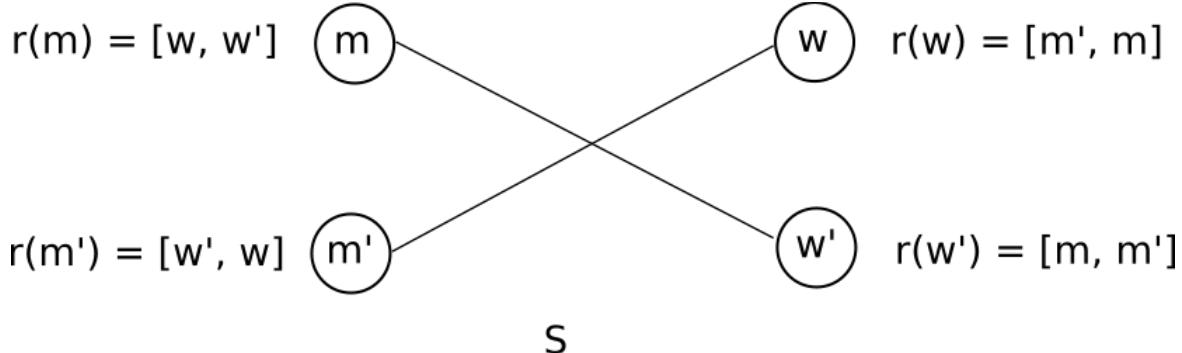
E se S for assim?



S é instável (tende a voltar para a configuração de cima). O que impede m e w de ficarem juntos no futuro? Instabilidade



Note que para os homens está OK (perfeito), mas para mulheres está trocado. Isso significa instabilidade? Não, pois S é estável. Veja que a mulher w' deseja melhorar e pode romper unilateralmente com m' . Porém, m não se interessa por w' e a restada (m, w') nunca irá se formar. O mesmo vale para o caso de w romper unilateralmente com m . A aresta (m', w) nunca irá se formar pois m' não quer w . E o que dizer desse outro S ?



S também é estável. Porém agora satisfaz plenamente as mulheres. Da mesma forma que o caso anterior, não há como as arestas (m, w) ou (m', w') aparecerem e portanto não há instabilidade.

Conceito: conjunto dominante – é o conjunto que propõe a ligação (casamento), quem toma a iniciativa

Questionamentos:

- 1) $\exists S$ estável para $\forall r(m_i)$ e $\forall r(w_i)$? Ou seja, é garantido que vai haver um S estável, independente de quais forem as listas de preferências? SIM
- 2) Dados $r(m_i)$ e $r(w_i)$ $\forall i$, como construir S estável? Algoritmo de Gale-Shapley
- 3) S é único dado $r(m_i)$ e $r(w_i)$ $\forall i$ e um conjunto dominante? SIM

Essa é a grande propriedade do algoritmo de Gale-Shapley. O algoritmo GS não fornece vantagem para qual nó será o primeiro a escolher, não se pode barganhar. Independente da ordem que o conjunto dominante faz as escolhas, sempre resulta no mesmo emparelhamento. Do ponto de vista da economia, é o que proporciona a regulação de mercados.

Algoritmo de Gale-Shapley (conj. M dominante)

Enquanto $\exists m_i$ livre (que ainda não tentou $\forall w_i \in W$)

Seja m_i esse homem

Seja w_i a mulher mais bem rankeada em $r(m_i)$ (para a qual m_i ainda não propôs)

Se w_i está livre

(m_i, w_i) “engaged” (adiciona em S temporariamente)

Senão

// significa que $\exists (\bar{m}, w_i) \in S$

Se $P(w_i, \bar{m}, m_i)$

m_i continua livre (vai continuar tentando)

Senão

// significa que $P(w_i, m_i, \bar{m})$

(m_i, w_i) “engaged”

\bar{m} fica livre (vai tentar outras parceiras)

Propriedades do algoritmo:

i) Ponto de vista de $m_i \in M$: a cada troca sequencia de parceiras piora

Para que está no conjunto dominante, sempre que houver uma troca será para pior

ii) Ponto de vista de $w_i \in W$: a cada troca sequencia de parceiros melhora

Para quem está conjunto passivo, sempre que houver uma troca será para melhorar

iii) Complexidade

No pior caso, cada m_i propõe no máximo para n mulheres. Como existe um total de n homens, a complexidade do algoritmo é $O(n^2)$

iv) Corretude

O emparelhamento S retornado pelo GS sempre é estável

Ex: Suponha que num site de relacionamentos existam as seguintes listas de preferências entre um grupo de rapazes e garotas.

BOY	1	2	3	4	5
Adam	Beth	Amy	Diane	Ellen	Cara
Bill	Diane	Beth	Amy	Cara	Ellen
Carl	Beth	Ellen	Cara	Diane	Amy
Dan	Amy	Diane	Cara	Beth	Ellen
Eric	Beth	Diane	Amy	Ellen	Cara

Boys' Preferences

GIRL	1	2	3	4	5
Amy	Eric	Adam	Bill	Dan	Carl
Beth	Carl	Bill	Dan	Adam	Eric
Cara	Bill	Carl	Dan	Eric	Adam
Diane	Adam	Eric	Dan	Carl	Bill
Ellen	Dan	Bill	Eric	Carl	Adam

Girls' Preferences

Utilizando o algoritmo de Gale-Shapley encontre:

- a) um emparelhamento estável M sendo os rapazes o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.
 b) um emparelhamento estável M' sendo as garotas o conjunto dominante. Mostre a sequencia de propostas até o emparelhamento estável.

a) Solução

i	m	w	engaged
1	Adam	Beth	yes
2	Bill	Diane	yes
3	Carl	Beth	yes
4	Adam	Amy	yes
5	Dan	Amy	no

6	Dan	Diane	yes
7	Bill	Beth	no
8	Bill	Amy	no
9	Bill	Cara	yes
10	Eric	Beth	no
11	Eric	Diane	yes
12	Dan	Cara	no
13	Dan	Beth	no
14	Dan	Ellen	yes

$$S^{(0)} = \emptyset$$

$$S^{(1)} = \{Adam, Beth\}$$

$$S^{(2)} = \{Adam, Beth\}, \{Bill, Diane\}$$

$$S^{(3)} = \{Carl, Beth\}, \{Bill, Diane\}$$

$$S^{(4)} = \{Carl, Beth\}, \{Bill, Diane\}, \{Adam, Amy\}$$

$$S^{(5)} = \{Carl, Beth\}, \{Bill, Diane\}, \{Adam, Amy\}$$

$$S^{(6)} = \{Carl, Beth\}, \{Dan, Diane\}, \{Adam, Amy\}$$

$$S^{(7)} = \{Carl, Beth\}, \{Dan, Diane\}, \{Adam, Amy\}$$

$$S^{(8)} = \{Carl, Beth\}, \{Dan, Diane\}, \{Adam, Amy\}$$

$$S^{(9)} = \{Carl, Beth\}, \{Dan, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}$$

$$S^{(10)} = \{Carl, Beth\}, \{Dan, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}$$

$$S^{(11)} = \{Carl, Beth\}, \{Eric, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}$$

$$S^{(12)} = \{Carl, Beth\}, \{Eric, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}$$

$$S^{(13)} = \{Carl, Beth\}, \{Eric, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}$$

$$S^{(14)} = \{Carl, Beth\}, \{Eric, Diane\}, \{Adam, Amy\}, \{Bill, Cara\}, \{Dan, Ellen\}$$

b) Solução

i	w	m	engaged
1	Amy	Eric	yes
2	Beth	Carl	yes
3	Cara	Bill	yes
4	Diane	Adam	yes
5	Ellen	Dan	yes

$$S^{(0)} = \emptyset$$

$$S^{(1)} = \{Amy, Eric\}$$

$$S^{(2)} = \{Amy, Eric\}, \{Beth, Carl\}$$

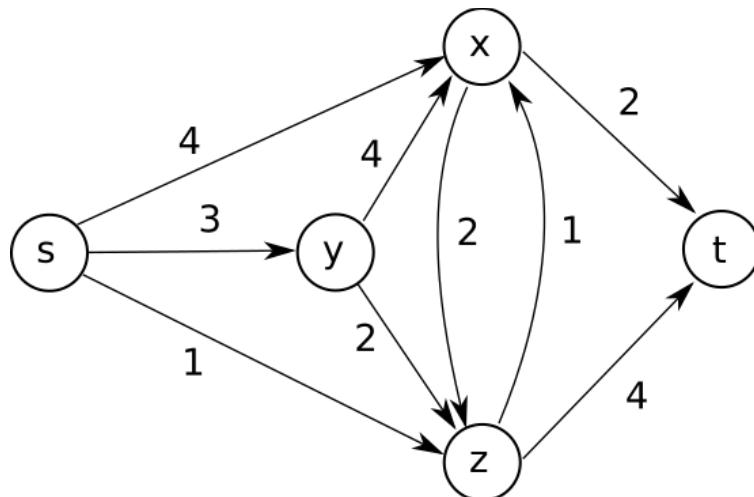
$$S^{(3)} = \{Amy, Eric\}, \{Beth, Carl\}, \{Cara, Bill\}$$

$$S^{(4)} = \{Amy, Eric\}, \{Beth, Carl\}, \{Cara, Bill\}, \{Diane, Adam\}$$

$$S^{(5)} = \{Amy, Eric\}, \{Beth, Carl\}, \{Cara, Bill\}, \{Diane, Adam\}, \{Ellen, Dan\}$$

Fluxo em redes

Motivação: Fornecedor s que deseja escoar seus produtos por diversos canais até chegar ao consumidor t



Consideraremos grafos direcionados $G = (V, E, c)$ onde $c: E \rightarrow N^+$ é a capacidade de uma aresta.
Iremos classificar os vértices de G como:

- $s \in V$: source (gerador de fluxo: não entra nada, apenas sai)
- $t \in V$: terminal ou sink (absorve fluxo: não sai nada, apenas entra)
- $\forall v \in V - \{s, t\}$: nós internos (intermediários)

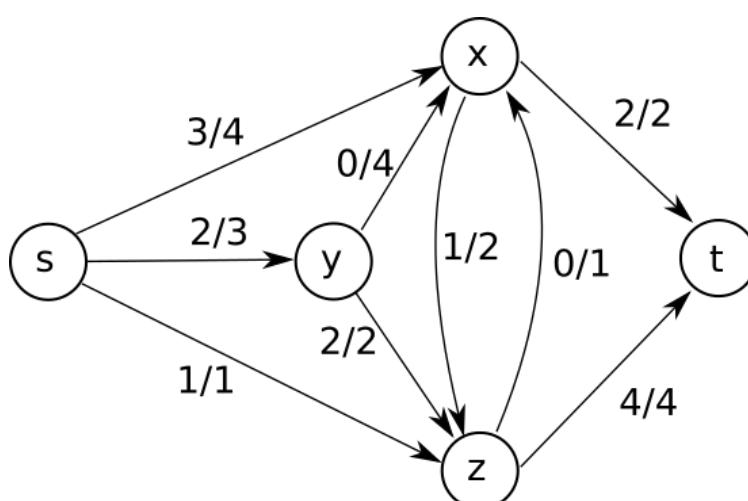
Em nossos exemplos iremos considerar apenas um único source e um único terminal

Def: Um fluxo $s-t$ é uma função $f: E \rightarrow N^+$ (associa um inteiro a cada aresta) que satisfaz:

- i) $\forall e \in E, f(e) \leq c(e)$ (restrição de capacidade)
- ii) $v(f) = \sum_{e \in O(s)} f(e) = \sum_{e \in I(t)} f(e)$ (fluxo gerado na fonte é igual ao fluxo consumido no terminal)
- iii) $\forall v \in V - \{s, t\}$ (nó interno)

$$\sum_{e \in I(v)} f(e) = \sum_{e \in O(v)} f(e) \quad (\text{conservação do fluxo})$$

Como exemplo, verifique que um fluxo válido para o grafo anterior é::



Basta verificar as 3 propriedades:

i) OK, pode-se ver facilmente que $\forall e \in E, f(e) \leq c(e)$

$$\text{ii)} \sum_{e \in O(s)} f(e) = 1+2+3=6=2+4=\sum_{e \in I(t)} f(e) \quad (\text{OK})$$

iii) Para $\{x, y, z\}$ temos

$$\sum_{e \in I(x)} f(e) = 3+0+0=1+2=\sum_{e \in O(x)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(y)} f(e) = 2=0+2=\sum_{e \in O(y)} f(e) \quad (\text{OK})$$

$$\sum_{e \in I(z)} f(e) = 1+1+2=0+4=\sum_{e \in O(z)} f(e) \quad (\text{OK})$$

Portanto, temos de fato um fluxo válido.

O Problema do Fluxo Máximo

Dado $G = (V, E, c)$ qual o máximo valor de $v(f)$ que pode chegar em t ?

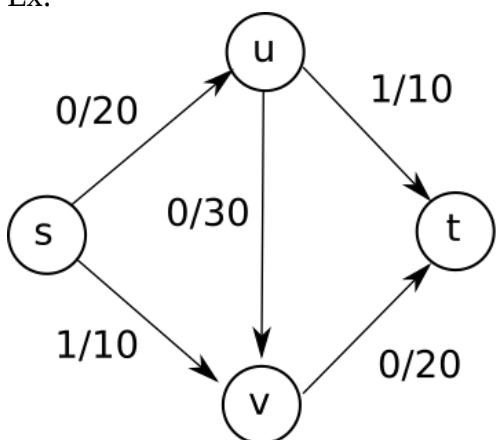
Ideia geral

Condição inicial: $f(e)=0, \forall e \in E$

Iteração: encontrar um caminho $s-t$ e transmitir fluxo

Condição de parada: Todo caminho $s-t$ encontra-se saturado

Ex:



Caso 1. $P = suvt, v(f) = 20$

Caso 2. $P_1 = sut, v(f) = 10$

$P_2 = svt, v(f) = 10 + 10 = 20$

$P_3 = suvt, v(f) = 20 + 10 = 30$

Note que em cada um dos casos, o valor do fluxo obtido é diferente. Não é bom que o valor do fluxo dependa da escolha dos caminhos, pois senão o algoritmo iria chegar a resultados diferentes para um mesmo problema. O ideal é que o fluxo máximo seja obtido independente da escolha dos caminhos. Para isso iremos definir o grafo residual.

Def: Grafo Residual $G_f = (V_f, E_f)$ gerado a partir de $G = (V, E, c)$

i) $V_f = V$ (conjunto de vértices é o mesmo)

ii) $E_f \neq E$ pois $\forall e \in E$ pode gerar até 2 arestas em E_f

a) Forward-Edge (FE): na mesma direção da aresta e

$\forall e \in E \text{ tal que } f(e) < c(e), \exists \text{ em } E_f \text{ uma aresta } e' \text{ com capacidade residual } c(e') = c(e) - f(e) \text{ (exatamente o que falta para saturar)}$

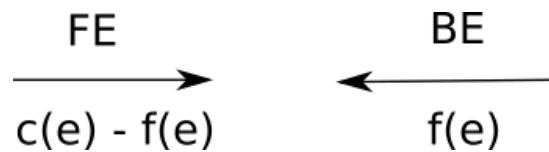
b) Backward-Edge (BE): na direção contrária a aresta e

$\forall e \in E \text{ tal que } f(e) > 0, \exists \text{ em } E_f \text{ uma aresta } e'' \text{ com capacidade residual } c(e'') = f(e) \text{ (exatamente o que já passa)}$

OBS: Note que no início não existem Backward-Edges pois os fluxos iniciais são nulos

RESUMO

1. $\forall e \in E \text{ com } f(e) = 0$ gera apenas Forward-Edge e'
2. $\forall e \in E \text{ com } f(e) = c(e)$ gera apenas Backward-Edge e''
3. $\forall e \in E \text{ com } 0 < f(e) < c(e)$ gera 2 arestas: e' (FE) e e'' (BE)



Para melhorar um fluxo inicial com valor f , devemos buscar por caminhos P_{st} no grafo residual $G_f = (V_f, E_f)$. A primitiva Augment que realiza essa operação de melhorar um fluxo f é definida como:

Augment(f, P) { (melhora fluxo f utilizando o caminho P em G_f)

$b = \text{gargalo}(P)$ (aresta de P com menor capacidade residual)

for each $e = (u, v)$ in P {

if e está a favor do fluxo $s-t$ em G

$f(e) = f(e) + b$

else

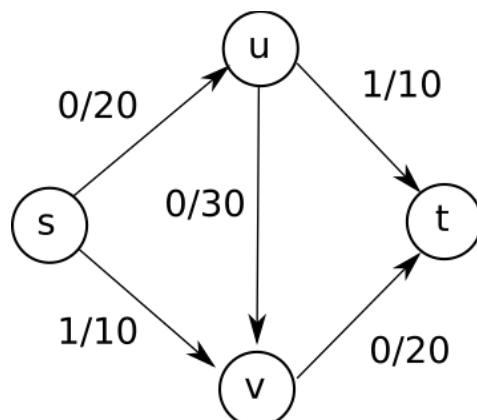
$f(e) = f(e) - b$

}

return f

}

Ex:



Passo 1: Grafo residual é o próprio G

$$f = 0$$

$$P = suvt$$

$$b = 20$$

$$f(su) = 0 + 20 = 20$$

$$f(uv) = 0 + 20 = 20$$

$$f(vt) = 0 + 20 = 20$$

Passo 2:

$$f = 20$$

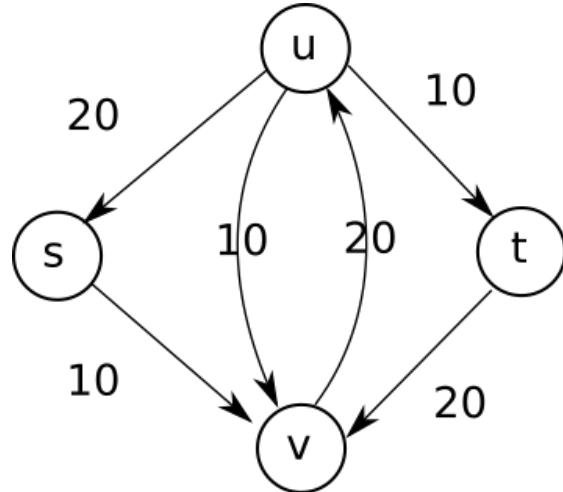
$$P = svut$$

$$b = 10$$

$$f(sv) = 0 + 10 = 10$$

Contrária a aresta (u,v)

$$f(vu) = f(uv) - 10 = 20 - 10 = 10$$

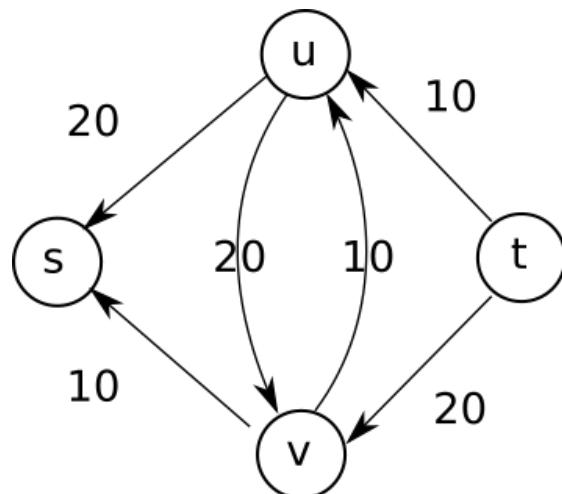


Passo 3:

$$f = 30$$

$\nexists P_{st}$ em $G_f \rightarrow$ PARE

O fluxo máximo vale 30

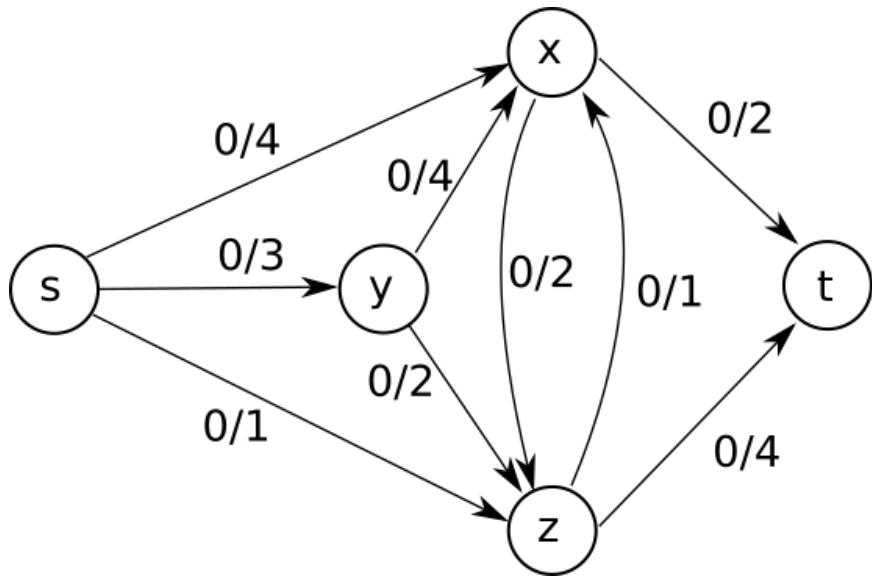


Essa sequência lógica de passos que foi realizada recebe o nome de algoritmo de Ford-Fulkerson

```
Max_Flow(G, s, t, c) {
    f = 0
    for each e in E
        f(e) = 0
    while ∃ Pst in Gf {
        Let Pst be one of these paths
        f' = Augment(f, Pst)
        Update the residual graph Gf
    }
}
```

Teorema: O fluxo retornado pelo algoritmo Ford-Fulkerson é máximo.

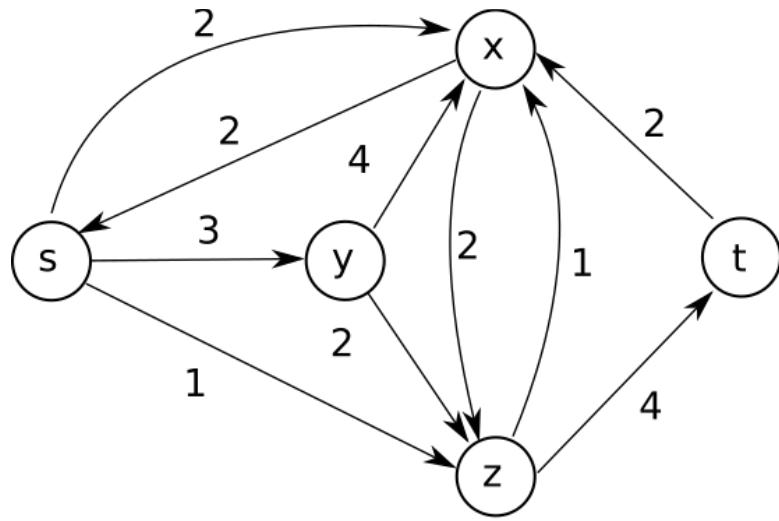
Ex: Utilizando ao algoritmo de Ford-Fulkerson, encontre o fluxo máximo



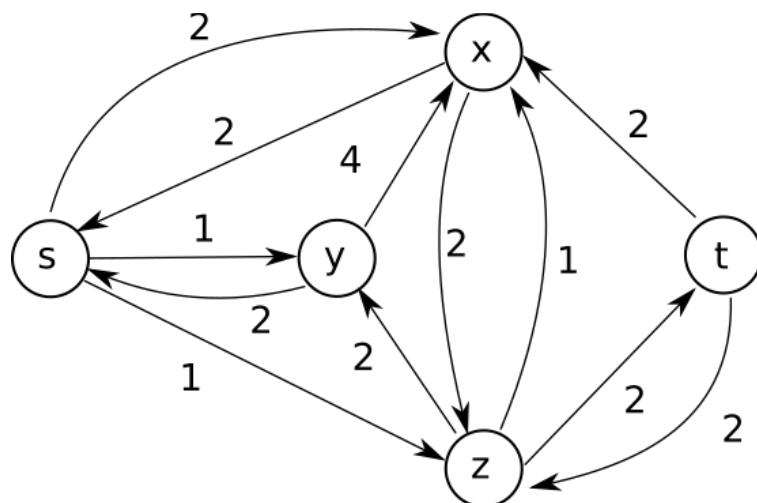
i	P_{st}	b	$f(e), \forall e \in P_{st}$	f
1	sxt	2	$f(sx) = 0 + 2 = 2$ $f(xt) = 0 + 2 = 2$	$0 + 2 = 2$
2	syzt	2	$f(sy) = 0 + 2 = 2$ $f(yz) = 0 + 2 = 2$ $f(zt) = 0 + 2 = 2$	$2 + 2 = 4$
3	szt	1	$f(sz) = 0 + 1 = 1$ $f(zt) = 2 + 1 = 3$	$4 + 1 = 5$
4	syxzt	1	$f(sy) = 2 + 1 = 3$ $f(yx) = 0 + 1 = 1$ $f(xz) = 0 + 1 = 1$ $f(zt) = 3 + 1 = 4$	$5 + 1 = 6$

Portanto, o fluxo máximo vale 6.

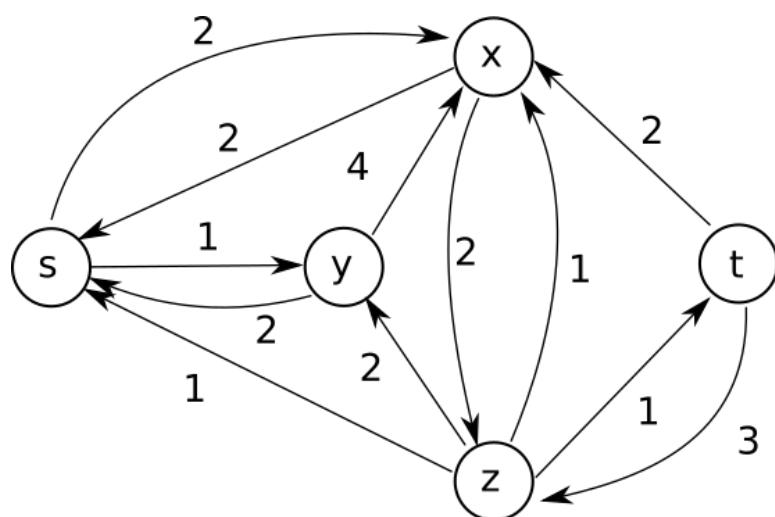
A seguir encontram-se os grafos residuais após cada um dos passos do algoritmo.



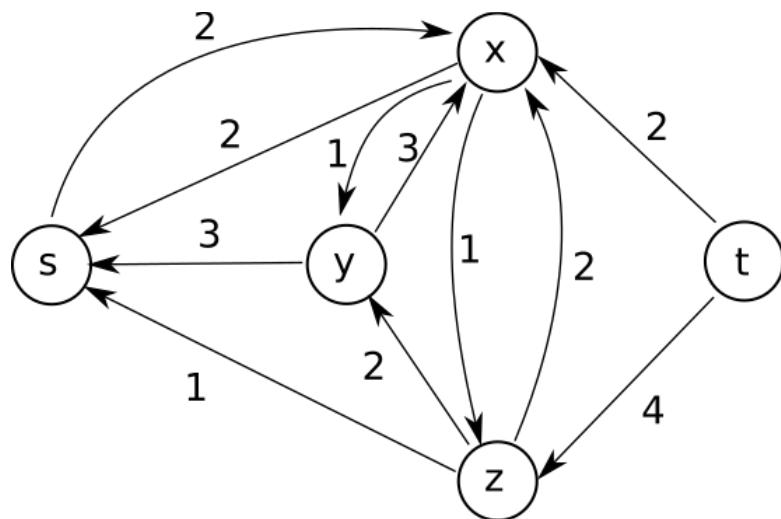
G_f Passo 1



G_f Passo 2

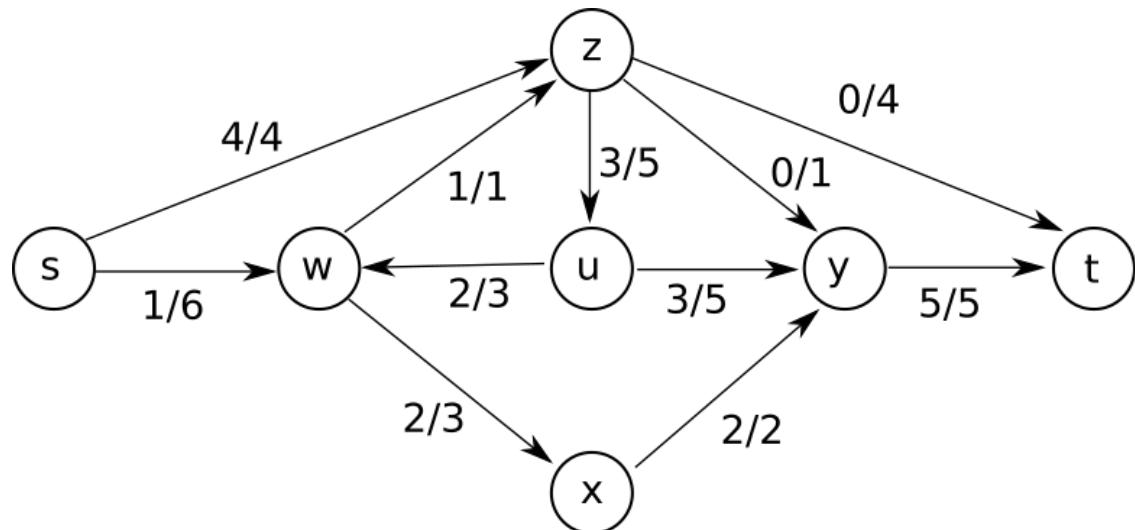


G_f Passo 3

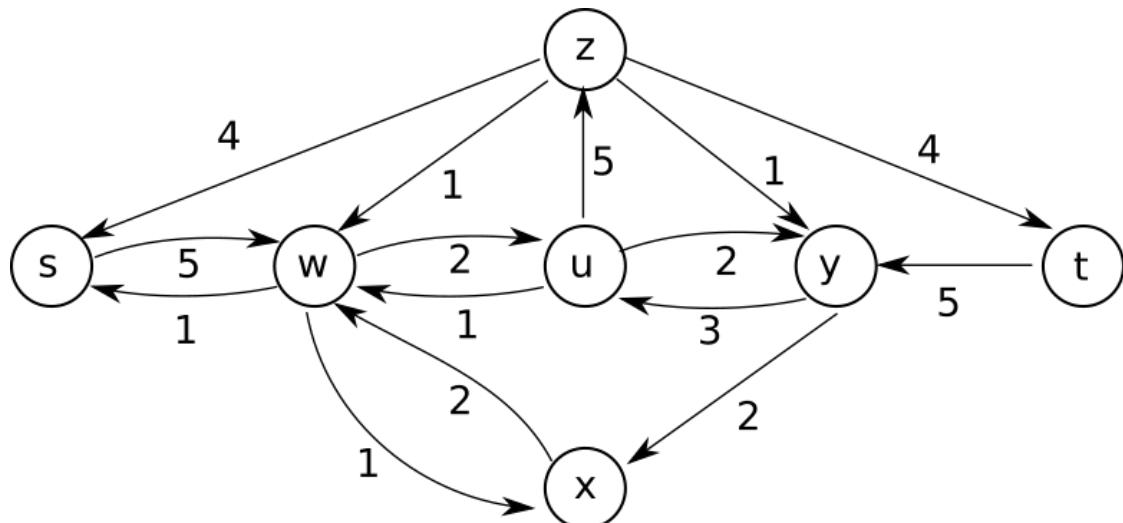


G_f Passo 4

Exercício: Dado o grafo G a seguir, responda: o fluxo dado é máximo? Justifique sua resposta.



Para verificar se o fluxo dado é máximo, devemos observar o grafo residual. Sendo assim, o grafo residual de G fica:



Note que $\exists P_{st} = swuzt$ no grafo residual, portanto o fluxo não pode ser máximo. Como o gargalo do caminho é 2, podemos aumentar o fluxo nessas arestas de duas unidades, ou seja:

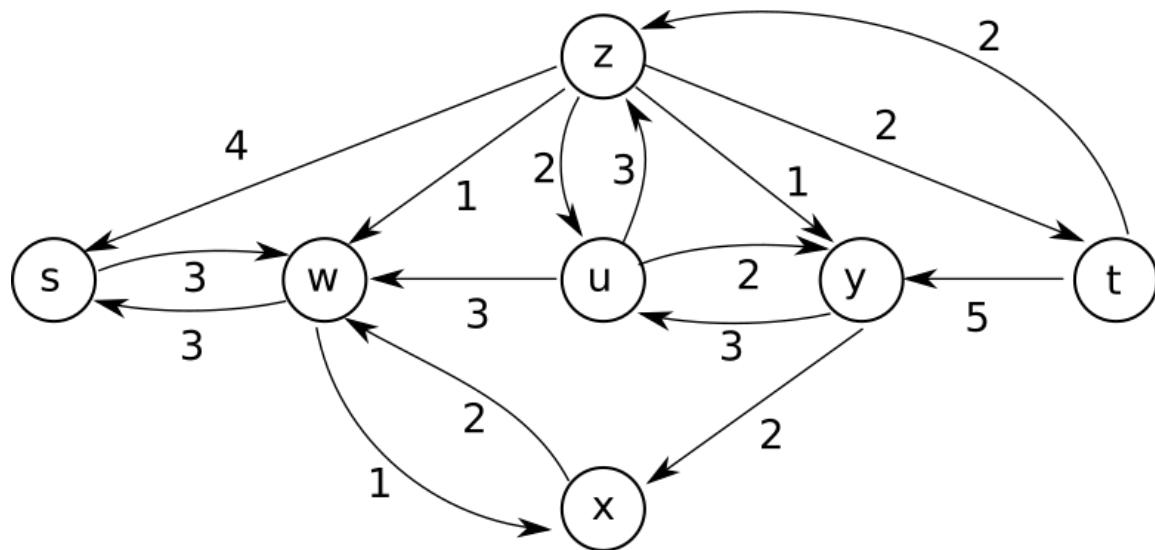
$$f(sw) = 1 + 2 = 3, \text{ pois a aresta } (s,w) \text{ está alinhada ao fluxo em } G$$

$$f(wu) = 2 - 2 = 0, \text{ pois a aresta } (w,u) \text{ está ao contrário do fluxo em } G$$

$$f(uz) = 5 - 2 = 3, \text{ pois a aresta } (u,z) \text{ está ao contrário do fluxo em } G$$

$$f(zt) = 0 + 2 = 2, \text{ pois a aresta } (z,t) \text{ está alinhada ao fluxo em } G$$

De modo que o valor total do fluxo agora é 7. O grafo residual é atualizado para:



Note que $\nexists P_{st}$ no grafo residual. Portanto, podemos concluir agora que o fluxo é máximo.