



Departamento de
Computação - **UFSCar**



Departamento de Computação
Centro de Ciências Exatas e Tecnologia
Universidade Federal de São Carlos

Teoria dos Grafos para Computação

Fundamentos teóricos, problemas, algoritmos e aplicações

Prof. Alexandre Luis Magalhães Levada
Email: alexandre.levada@ufscar.br

Sumário

- 1. Fundamentos básicos**
- 2. O problema do isomorfismo**
- 3. Cadeias de Markov e random walks**
- 4. O modelo Pagerank**
- 5. Árvores**
- 6. O Problema da árvore geradora mínima**
 - O algoritmo de Kruskal
 - O algoritmo de Prim
- 7. Busca em grafos**
 - Busca em Largura (Breadth-First Search)
 - Busca em Profundidade (Depth-First Search)
- 8. Caminhos mínimos em grafos**
 - O algoritmo de Bellman-Ford
 - O algoritmo de Djikstra
 - A heurística A*
- 9. Grafos Eulerianos**
 - O problema do carteiro chinês
- 10. Grafos Hamiltonianos**
 - O problema do caixeiro viajante
- 11. Grafos Planares**
- 12. Coloração de vértices**
- 13. Emparelhamentos**
 - O algoritmo húngaro
- 14. O problema do fluxo máximo em redes**
 - O algoritmo Ford-Fulkerson
 - O Teorema Min-cut/Max-flow
- 15. Introdução as redes complexas**

"Solutions are not found by pointing fingers; they are reached by extending hands."
-- Aysha Taryam

Prólogo

A teoria dos grafos é um dos pilares fundamentais da ciência da computação moderna, fornecendo as bases teóricas e práticas para modelar e resolver uma ampla gama de problemas. Esta apostila foi elaborada com o intuito de oferecer aos estudantes uma visão abrangente e aplicada dos principais conceitos, algoritmos e aplicações de grafos, com foco especial no contexto computacional. A seguir, apresentamos um panorama dos capítulos que compõem o material, para orientar o leitor na trajetória de aprendizagem e motivar a exploração dos temas abordados.

*Capítulo 1 – Fundamentos Básicos: Introduz os conceitos elementares de grafos, como vértices, arestas, graus, subgrafos, grafos bipartidos e completos, além de representações computacionais. É o alicerce para todos os demais capítulos, ideal para formar uma base sólida.

*Capítulo 2 – O Problema do Isomorfismo: Explora o desafio de determinar se dois grafos são estruturalmente equivalentes. O capítulo destaca propriedades invariantes e a complexidade computacional envolvida, com exemplos práticos e exercícios instigantes.

*Capítulo 3 – Cadeias de Markov e Caminhadas Aleatórias: Apresenta os fundamentos das cadeias de Markov e sua aplicação em caminhadas aleatórias sobre grafos, fornecendo uma ponte entre teoria dos grafos e processos estocásticos.

*Capítulo 4 – O Modelo PageRank: Analisa o famoso algoritmo de ranqueamento de páginas web, baseado em caminhadas aleatórias em dígrafos com fator de amortecimento. O modelo é apresentado de forma detalhada e com base matemática sólida.

*Capítulo 5 – Árvores: Discute propriedades das árvores e sua importância em diversos contextos computacionais. Inclui teoremas clássicos, caracterizações, e motivações combinatórias como o Código de Prüfer e o Teorema de Cayley.

*Capítulo 6 – O Problema da Árvore Geradora Mínima: Trata da tarefa de encontrar uma árvore geradora com peso mínimo em um grafo ponderado, essencial em redes e sistemas de comunicação. Explica detalhadamente o algoritmo de Kruskal para a construção de árvores geradoras mínimas, com base em ordenação e estruturas de conjuntos disjuntos. Apresenta uma abordagem alternativa e eficiente ao problema da árvore geradora mínima, o algoritmo de Prim, baseada em expansão progressiva da árvore a partir de um vértice inicial.

*Capítulo 7 – Busca em Grafos: Introduz os algoritmos de busca em largura (BFS) e profundidade (DFS), ferramentas essenciais para exploração e análise de grafos, incluindo construção de árvores de busca e marcação de componentes conexos.

Capítulo 8 – Caminhos Mínimos em Grafos: Aborda algoritmos para encontrar os caminhos mais curtos entre vértices, fundamentais em roteamento, jogos e otimização. Inclui Bellman-Ford, Dijkstra e a heurística A.

*Capítulo 9 – Grafos Eulerianos e o Problema do Carteiro Chinês: Explora condições para existência de ciclos e trilhas eulerianas, e como aplicá-las na resolução de problemas de roteamento com cobertura mínima de arestas.

*Capítulo 10 – Grafos Hamiltonianos e o Problema do Caixeiro Viajante: Discute a existência de ciclos que visitam todos os vértices, e introduz o TSP, um problema clássico de otimização combinatória e difícil resolução.

*Capítulos 11 e 12 – Grafos Planares e Coloração de Vértices: Analisa propriedades de grafos que podem ser desenhados no plano sem cruzamento de arestas, e introduz o problema da coloração, com aplicações em alocação de recursos.

*Capítulo 13 – Emparelhamentos em Grafos: Estuda como formar pares de vértices sem sobreposição de arestas, incluindo o algoritmo húngaro.

*Capítulo 14 – Fluxo Máximo em Redes: Apresenta problemas de fluxo em redes direcionadas e capacidades, abordando os algoritmos de Ford-Fulkerson e o teorema min-cut/max-flow, essenciais em logística e engenharia de redes.

*Capítulo 15 – Introdução às redes complexas: Finaliza com uma discussão introdutória sobre redes complexas e seus principais modelos: Erdös-Renyi, Watts-Strogatz e Barabasi-Alberts.

Este material foi cuidadosamente estruturado para combinar clareza teórica com aplicabilidade prática. Ao longo da leitura, o estudante será desafiado com demonstrações matemáticas, algoritmos clássicos e problemas que ilustram a importância dos grafos no mundo real. Espera-se que essa jornada pelo universo da teoria dos grafos e suas aplicações inspire não apenas o domínio técnico, mas também o encantamento com a elegância e versatilidade dessa área tão rica da ciência da computação.

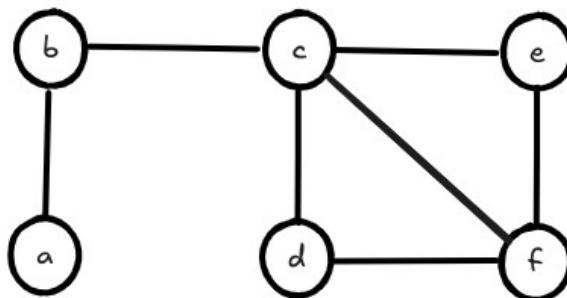
Grafos: Fundamentos Básicos

Grafos são objetos matemáticos que representam relações binárias entre elementos de um conjunto finito. Eles são utilizados na computação para representar estruturas complexas em que cada nó pode ser conectar a um número variável de vizinhos, como por exemplo, a internet e as redes sociais. Em termos gerais, um grafo consiste em um conjunto de vértices que podem estar ligados dois a dois por arestas. Se dois vértices são unidos por uma aresta, então eles são vizinhos. É uma estrutura fundamental para a computação, uma vez que diversos problemas do mundo real podem ser modelados com grafos, como encontrar caminhos mínimos entre dois pontos, alocação de recursos e modelagem de redes complexas.

Def: $G = (V, E)$ 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: conjunto de **arestas**

A figura a seguir ilustra um grafo $G = (V, E)$ arbitrário.



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

Def: 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: $d(a) = 1$, $d(b) = 2$, $d(c) = 4$, , ...

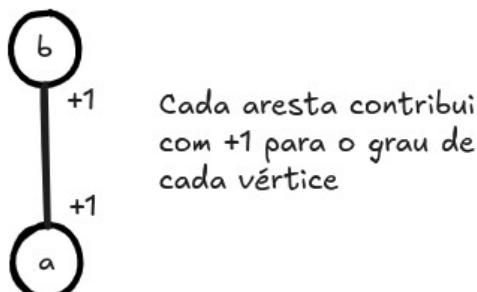
Def: A lista de graus de $G = (V, E)$ é a lista que armazena os graus dos vértices em ordem crescente.

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

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.



Prova por indução: Seja $P(n)$ definido como:

$$P(n): \sum_{i=1}^n d(v_i) = 2m$$

=> BASE: Note que nesse caso, $n = 1$, o que implica dizer que temos um único vértice. Então, para todo $m \geq 0$ (número de arestas), a soma dos graus será sempre um número par pois ambas as extremidades das arestas incidem sobre o único vértice de G .

=> PASSO DE INDUÇÃO: Para k arbitrário, mostrar que $P(k) \rightarrow P(k+1)$

Note que para k vértices, temos:

$$P(k): \sum_{i=1}^k d(v_i) = 2m$$

Ao adicionarmos exatamente um vértice a mais, temos $k+1$ vértices:

$$P(k+1): \sum_{i=1}^{k+1} d(v_i) = 2m'$$

Ao passar de k para $k+1$ vértices, temos duas opções:

- a) o grau do novo vértice é zero;
- b) o grau do novo vértice é maior que zero.

Caso a): Nessa situação, temos o número de arestas permanece inalterado, ou seja, $m = m'$. Pela hipótese de indução e sabendo que o grau no novo vértice é zero, podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^k d(v_i) + d(v_{k+1}) = 2m + 0 = 2m'$$

ou seja, $P(k+1)$ é válida.

Caso b): Nessa situação, temos que o número de arestas $m' > m$. Seja $m' = m + a$, onde a denota o número de arestas adicionadas ao inserir o novo vértice $k+1$. Então, pela hipótese de indução e sabendo que o grau do novo vértice será a , podemos escrever:

$$\sum_{i=1}^{k+1} d(v_i) = \sum_{i=1}^k d(v_i) + d(v_{k+1}) + 1 + 1 + 1 + \dots + 1$$

a vezes 1

pois para cada extremidade das arestas no novo vértice, haverá outra extremidade em algum outro vértice. Isso implica em:

$$\sum_{i=1}^{k+1} d(v_i) = 2m + a + a = 2m + 2a = 2(m + a) = 2m'$$

ou seja, $P(k+1)$ é válida. Note que mesmo que alguma das arestas inseridas possuam ambas as extremidades no novo vértice $k+1$, a soma dos graus também será igual a $2m + 2a$, o que continuará validando $P(k+1)$. Portanto, a prova está concluída.

Teorema: Em um grafo $G = (V, E)$ o número de vértices com grau ímpar é sempre par.

Podemos partitionar V em 2 conjuntos: P (grau par) e I (grau ímpar). Assim,

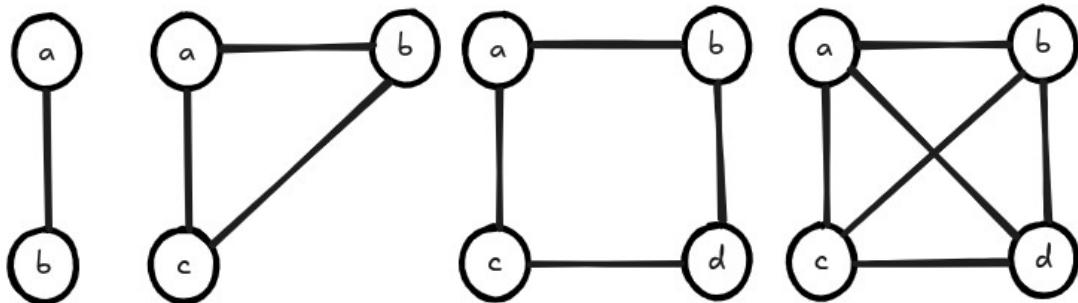
$$\sum_{i=1}^n d(v_i) = \sum_{v \in P} d(v) + \sum_{u \in I} d(u) = 2m$$

Isso implica em

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

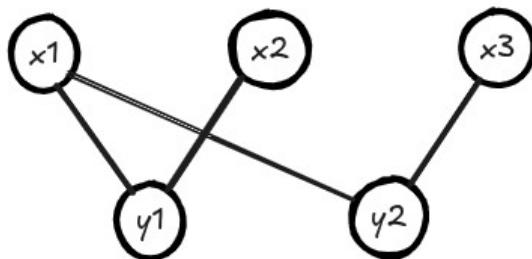
Como $2m$ é par e a soma de números pares é sempre par, resulta que a soma dos números ímpares também é par. Para que isso ocorra temos que ter $|I|$ par (número de elementos do conjunto I é par).

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

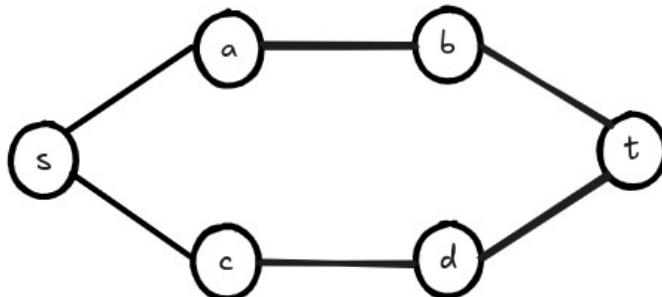


Def: Grafo Bipartido

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



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



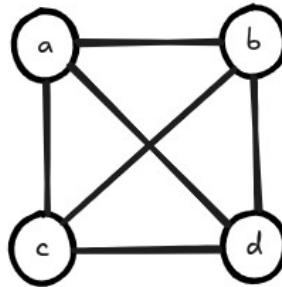
SIM, é bipartido! Mesmo parecendo que não.

Como decidir se grafo G é bipartido? Seja $R = \{0, 1\}$ o conjunto de rótulos e seja $r \in R$ um rótulo arbitrário e $\bar{r} \in R$ o seu complementar, ou seja, $\bar{r} = 1 - r$.

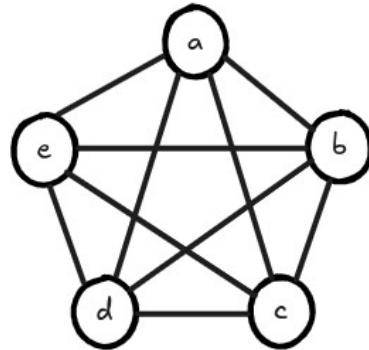
1. Escolha um vértice inicial v e rotule-o como r .
2. Para todos os vértices u vizinhos de v ainda não rotulados, rotule-os como \bar{r} . Ou seja, se um vértice recebe rótulo r , todos seus vizinhos não rotulados devem receber rótulo \bar{r} e vice-versa.
3. Condição de parada: Pare quando todos os vértices do grafo estiverem rotulados.
4. Se ao fim do processo toda aresta do grafo for do tipo (r, \bar{r}) então o grafo G 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 $L_G = (n-1, n-1, n-1, \dots, n-1)$



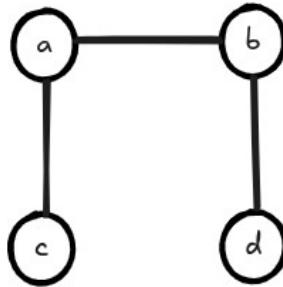
K_4



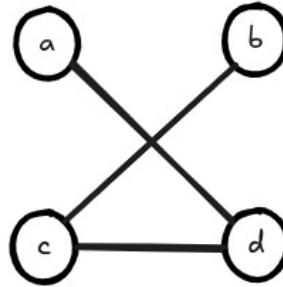
K_5

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

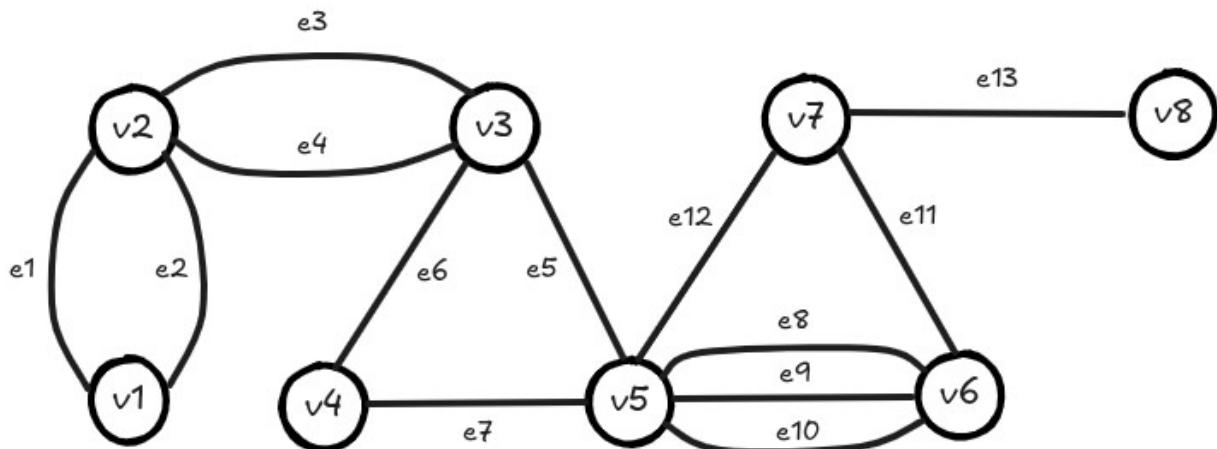
Seja $G = (V, E)$ um grafo. Dizemos que $H = (V', E')$ é um subgrafo de G se $V' \subseteq V$ e $E' \subseteq E$. Em outras palavras, é todo grafo que pode ser obtido a partir de G através de remoção de vértices e/ou arestas.

Def: Subgrafos disjuntos não possuem vértices em comum

Subgrafos arestas disjuntos: não possuem aresta em comum

Caminhos e ciclos

Passeios, trilhas e caminhos são conceitos fundamentais na teoria dos grafos e desempenham um papel central na modelagem e análise de diversas estruturas e sistemas complexos. Um **passeio** em um grafo é uma sequência de vértices e arestas onde cada aresta conecta dois vértices consecutivos na sequência; ele pode passar repetidamente pelos mesmos vértices e arestas. Já uma **trilha** é um tipo especial de passeio em que nenhuma aresta é repetida, embora os vértices possam ser. Por fim, um **caminho** é uma trilha ainda mais restrita, em que todos os vértices são distintos, o que implica também a não repetição de arestas. Essas distinções são essenciais para o estudo de conectividade, acessibilidade, otimização e outras propriedades estruturais em grafos, com aplicações que vão desde redes de transporte e circuitos elétricos até algoritmos de busca e análise de redes sociais. A figura a seguir ilustra alguns exemplos.



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.

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.

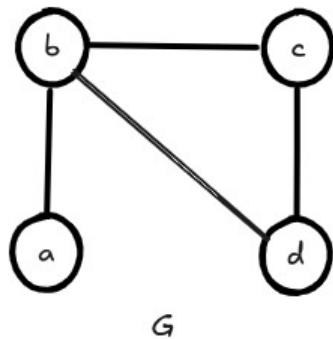
O caminho/trilha/passeio trivial é aquele composto por zero arestas

Representações computacionais de grafos

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

a) Grafos básicos simples

$$A_{i,j} = \begin{cases} 1, & j \in N(i) \\ 0, & j \notin N(i) \end{cases}$$



$$A = \begin{bmatrix} a & b & c & d \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \begin{array}{l} a \\ b \\ c \\ d \end{array}$$

Propriedades básicas

i) $\text{diag}(A) = 0$

ii) matriz binária

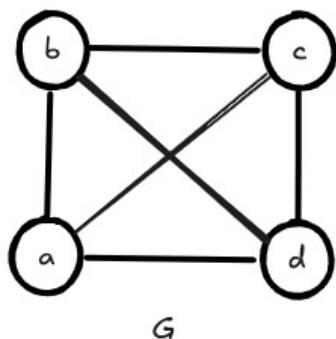
iii) $A = A^T$ (com exceção de grafos direcionados)

iv) $\sum_j A_{i,j} = d(v_i)$

v) Esparsa

vi) $O(n^2)$ em espaço – requer $\frac{n^2}{8}$ bytes para armazenamento (contíguos na memória)

2. Matriz de Incidência M: matriz n x m em que as linhas referem-se aos vértices e as colunas referem-se as arestas:

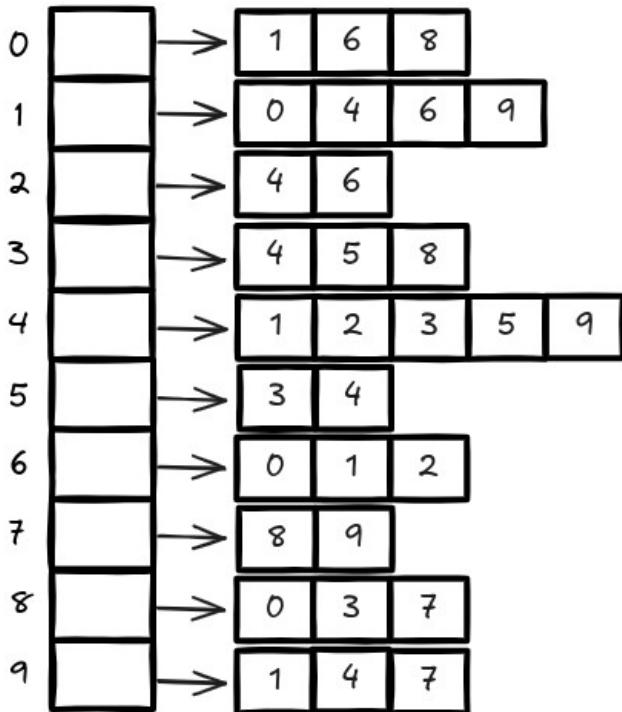


$$M = \begin{bmatrix} e_1 & e_2 & e_3 & e_4 & e_5 & e_6 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \begin{array}{l} a \\ b \\ c \\ d \end{array}$$

$\Rightarrow O(nm)$ em espaço – requer $\frac{nm}{8}$ bytes para armazenamento (contíguos na memória)

3. Lista de adjacências: estrutura dinâmica em que cada nó possui uma referência para uma lista encadeada com seus vizinhos.

Lista de adjacências



Node
int vertex
Node prox

AdjList
Node array L[n]

Em resumo, a lista de adjacências é um array de tamanho n de referências para Node.

Note que ao contarmos a soma dos graus dos vértices (somatório dos tamanhos de cada lista encadeada), estamos de fato contando cada aresta duas vezes. Assim, de acordo com o *Handshaking Lema*, devemos dividir esse número total por 2 para obter o número exato de arestas.

Como temos n referências e cada referência é uma lista encadeada possui $d(v_i)$ nós, temos que o número total de nós é igual a $\sum_{i=1}^n d(v_i) = 2m$, o que resulta em $O(m)$.

Pergunta: Quantos bytes são necessários para armazenar um grafo G em memória com uma lista de adjacências?

Supondo que cada inteiro ocupa 32 bits, temos que cada nó ocupa $2m \left(\frac{32}{8}\right)$ bytes, o que equivale a 8m bytes (não precisa ser contíguo na memória).

Sabendo que a densidade de um grafo é dada por $d = \frac{m}{n^2}$, quando é preferível usar uma lista de adjacências em detrimento a uma matriz de adjacências?

A lista de adjacências deve ser escolhida se:

$$8m < \frac{n^2}{8} \rightarrow \frac{m}{n^2} < \frac{1}{64} \rightarrow d < \frac{1}{64} \rightarrow d < 0.015625 \text{ (grafo esparsa)}$$

Ex: Se $n = 100$ e $m = 150$, temos $d = 150/10000 = 0.015$, o que é menor que 0.01562

Lembrando que o máximo valor de densidade é dado por:

$$d = \frac{n(n-1)}{2n^2} = \frac{n^2-n}{2n^2} = \frac{1}{2} - \frac{1}{2n} = \frac{1}{2} \left(1 - \frac{1}{n}\right)$$

o que tende a 0.5 quando n cresce arbitrariamente.

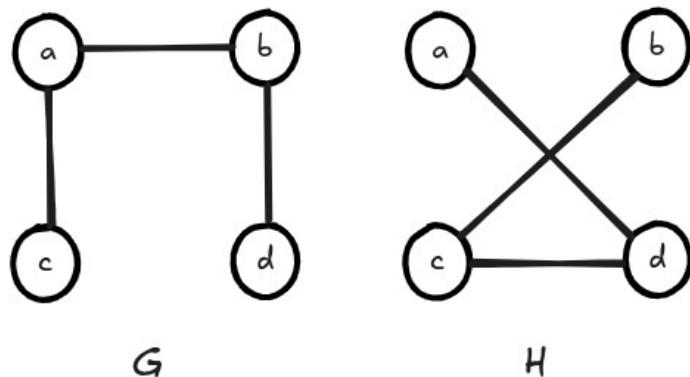
Na linguagem Python, uma excelente biblioteca para se trabalhar com grafos é a NetworkX. Um guia de referência oficial para a essa biblioteca pode ser encontrada em:

https://networkx.github.io/documentation/stable/_downloads/networkx_reference.pdf

O leitor poderá encontrar diversos exemplos práticos de inúmeras funções da biblioteca.

O Problema do Isomorfismo

O problema do isomorfismo em grafos é uma questão central na teoria dos grafos que busca determinar se dois grafos são estruturalmente iguais, independentemente da forma como seus vértices estão rotulados. Dois grafos são considerados **isomorfos** se existe uma correspondência bijetiva entre seus conjuntos de vértices que preserve as adjacências, ou seja, se dois vértices estão conectados por uma aresta em um grafo, então seus correspondentes também estão conectados no outro. Em termos práticos, isso significa que os dois grafos possuem a mesma estrutura, embora possam parecer diferentes à primeira vista devido à disposição ou rotulagem dos vértices. Resolver o problema do isomorfismo é particularmente desafiador porque não há, até hoje, um algoritmo eficiente conhecido que o resolva em tempo polinomial para todos os casos, nem se sabe se ele pertence à classe NP-completo. Esse problema tem implicações importantes em áreas como reconhecimento de padrões, química computacional, análise de redes e segurança da informação.

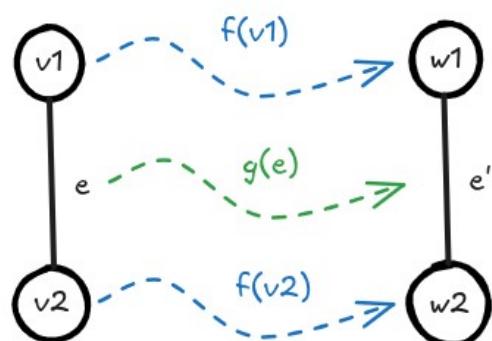


Def: $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ são isomorfos se:

- 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

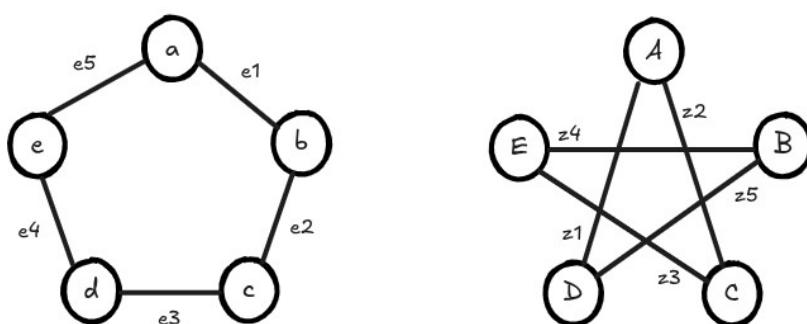
satisfazendo a seguinte restrição (*)

$$v_1 \sim_e v_2 \Leftrightarrow f(v_1) \sim_{g(e)} f(v_2)$$



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

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



Passo 1: Encontrar mapeamento entre os vértices f

v	a	b	c	d	e
f(v)	A	C	E	B	D

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

e	e1	e2	e3	e4	e5
g(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 $f(a)$: $(D, A) = z1$

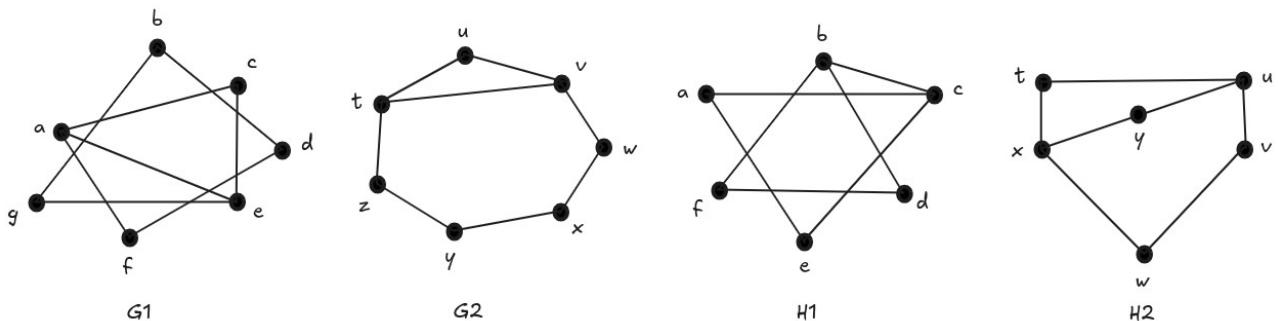
Portanto, os grafos G_1 e G_2 são isomorfos.

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,

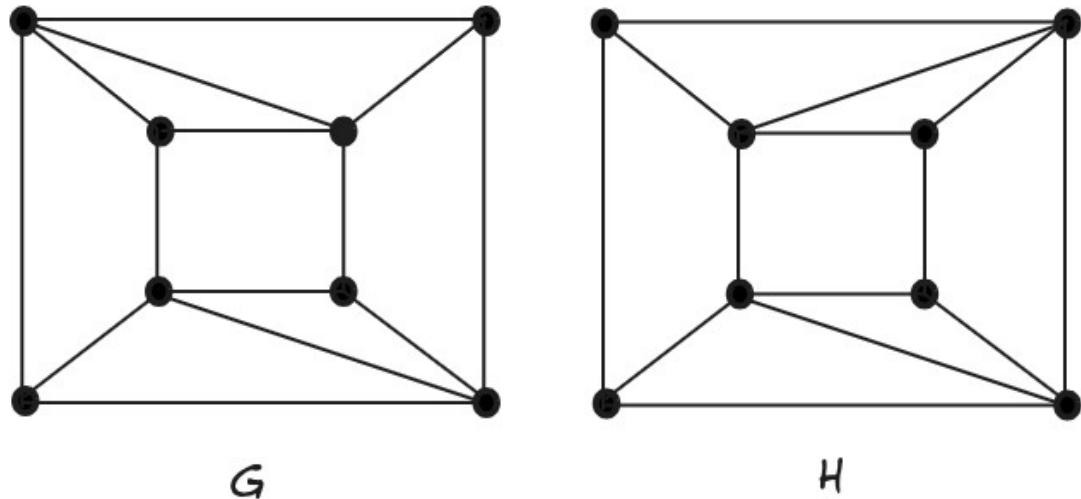
- 1) $|V_1| = |V_2|$ (o número de vértices é igual)
- 2) $|E_1| = |E_2|$ (o número de arestas é igual)
- 3) $\omega(G_1) = \omega(G_2)$ (mesmo número de componentes conexos)
- 4) $L_{G_1} = L_{G_2}$ (listas de graus são idênticas)
- 5) Ambos G_1 e G_2 admitem ciclos de comprimento k , para $k \leq n$

Ex: Quais dos pares a seguir são isomorfos? Prove ou refute.



Note que G_1 é isomorfo a G_2 (prove isso, encontrando os mapeamentos f e g). Porém, H_1 não é isomorfo a H_2 pois enquanto H_1 admite ciclo de comprimento 3, H_2 não admite.

Ex: Os grafos a seguir são isomorfos ou não? Prove sua resposta.



Ambos possuem o mesmo número de vértices: 8

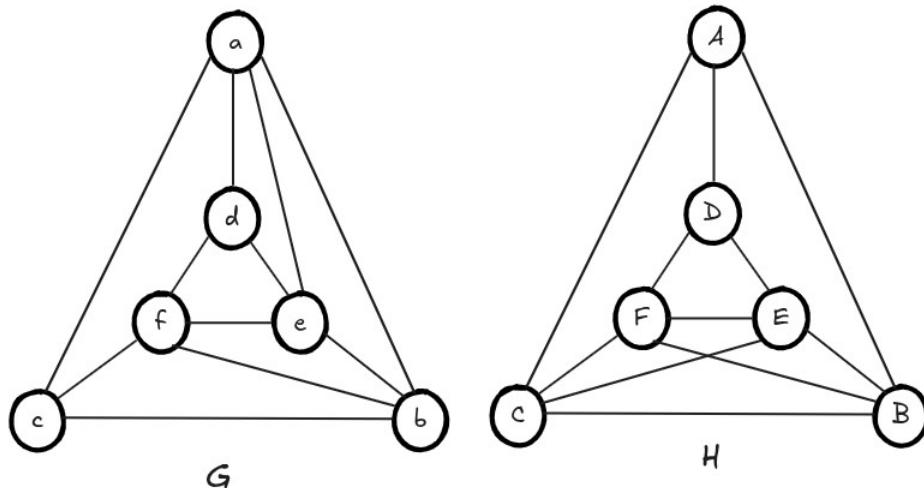
Ambos possuem o mesmo número de arestas: 8

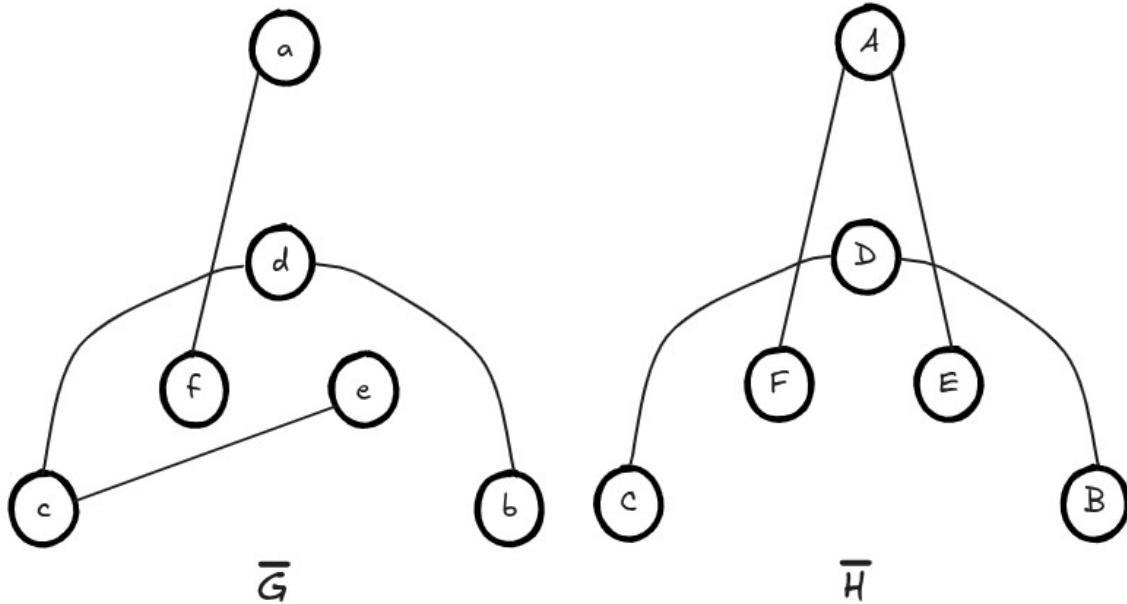
Ambos possuem a mesma lista de graus: (3, 3, 3, 3, 4, 4, 4, 4)

Ambos possuem ciclos de comprimentos 3, 4, 5, 6, 7 e 8

A princípio, parece que G é isomorfo a H . Porém, isso não é verdade. Note que, 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 G , todos os vértices de grau 4 possuem como vizinhos exatamente um único outro vértice de grau 4, enquanto em H , cada vértice de grau 4 possui exatamente 2 vértices de grau 4 como vizinhos. Isso significa uma ruptura em uma aresta, o que altera a topologia (você consegue identificar qual aresta foi cortada e religada?)

Teorema: Dois grafos são isomorfos se e somente se seus complementares também forem, ou seja, $G_1 \equiv G_2 \Leftrightarrow \bar{G}_1 \equiv \bar{G}_2$

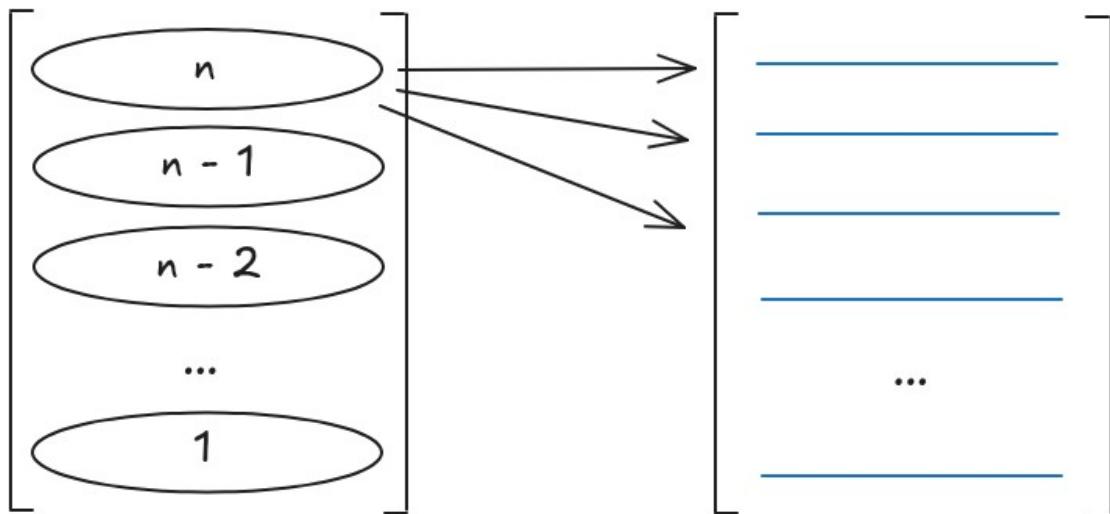




Note que no primeiro grafo os vértices de grau 2 são adjacentes, o que não ocorre no segundo.

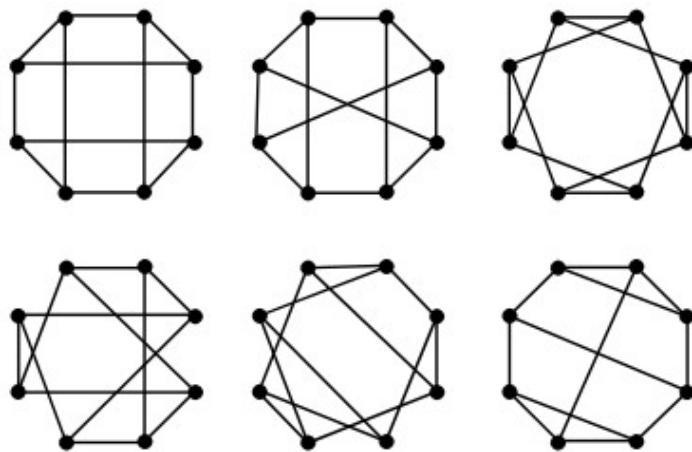
Teorema: $G_1 \equiv G_2 \Leftrightarrow A_1 = A_2$ para alguma sequencia de permutações de linhas e colunas.

Dadas duas matrizes de adjacências A_1 e A_2 , devemos permutar linhas e colunas de A_1 de modo a torná-la idêntica a A_2 . Porém, essa metodologia tem um sério problema. Note que, em termos de complexidade, o número de permutações totais para as linhas é $n!$

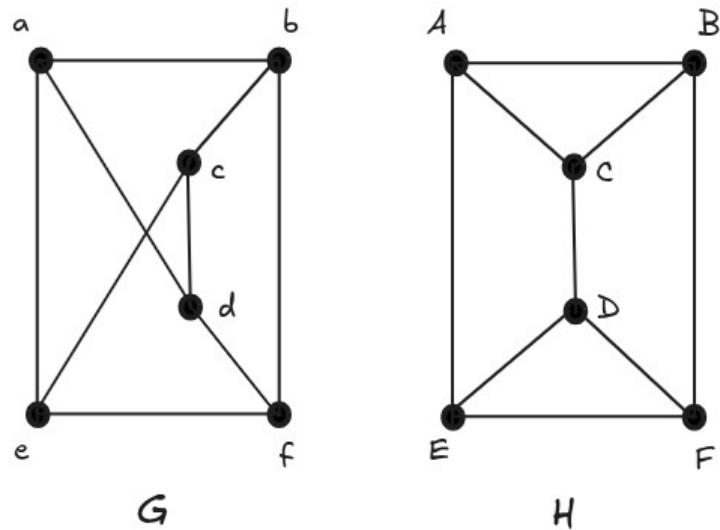


O mesmo vale para as colunas, de forma que a complexidade do algoritmo é da ordem de $O(n!)$, tornando o método inviável para a maioria dos grafos. Atualmente, ainda não se conhecem algoritmos polinomiais para esse problema. Para algumas classes de grafos o problema é polinomial (árvores, grafos planares).

Ex: Os seis grafos a seguir consistem de objetos isomorfos. Identifique quais são isomorfos a quais.

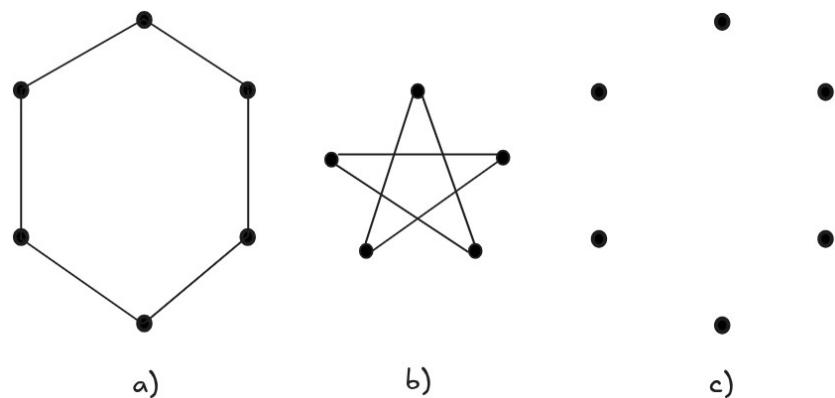


Ex: Os dois grafos a seguir são isomorfos ou não? Prove sua resposta.



Ex: Um grafo simples G é chamado de autocomplementar se ele for isomorfo ao seu próprio complemento.

a) Quais dos grafos a seguir são autocomplementares?



b) Prove que, se G é um grafo autocomplementar com n vértices, então $n = 4t$ ou $n = 4t + 1$, para algum inteiro t (dica: considere o número de arestas do K_n).

Conectividade em grafos

A conectividade em grafos é um conceito fundamental que descreve o grau de interligação entre os vértices de um grafo, sendo essencial para entender sua estrutura e funcionalidade. Um grafo é dito **conexo** se existe pelo menos um caminho entre qualquer par de vértices, o que garante que nenhuma parte do grafo está isolada. Em contrapartida, um grafo **desconexo** possui vértices ou subconjuntos de vértices que não se comunicam entre si. No caso de grafos dirigidos, distingue-se entre conectividade **forte**, quando há um caminho em ambas as direções entre todos os pares de vértices, e conectividade **fraca**, quando tal propriedade só vale se ignorarmos a direção das arestas. A análise da conectividade permite identificar componentes conectados, avaliar a robustez de redes, detectar pontos de falha crítica (como vértices ou arestas cuja remoção desconecta o grafo), e tem aplicações em áreas como redes de computadores, sistemas de transporte, biologia computacional e análise de redes sociais. 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: $G = (V, E)$ é conexo se e somente se $\forall u, v \in V (\exists \text{ caminho } P_{uv})$

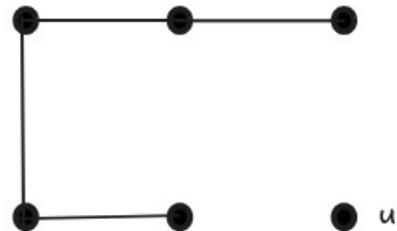
Def: Componente conexo: subgrafo conexo máximo

Teorema: Seja $G = (V, E)$ um grafo conexo. Então, $|E| \geq |V| - 1$

1. Pela contrapositiva, temos: Se $|E| < |V| - 1$, então G é desconexo.
2. Suponha um grafo G , com n vértices e menos de $n-1$ arestas.
3. Podemos notar que não há como G ser conexo, uma vez que sempre haverá um vértice u isolado.



u



$n = 6$
 $m = 4$

Teorema: Seja $G = (V, E)$ um grafo. Se $|E| > \binom{|V|-1}{2}$, então o grafo G é conexo.

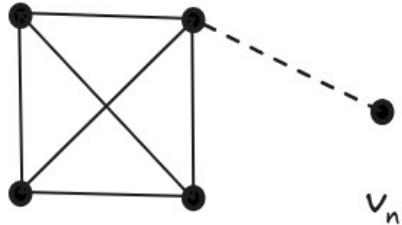
Pela contrapositiva, temos: Se G não é conexo, então $|E| \leq \binom{|V|-1}{2}$.

1. Suponha um grafo G desconexo. Então, existe um par de vértices $u, v \in V$ para o qual não existe um caminho P_{uv} .
2. Sendo assim, deve existir ao menos um vértice u isolado, mesmo que todos os outros $n - 1$ vértices sejam totalmente conectados.
3. Suponha que todos os vértices, com exceção de u , sejam totalmente conectados. Então, o subgrafo formado por eles define o K_{n-1} .

Sendo assim, o número de arestas em K_{n-1} é dado por $\binom{|V|-1}{2}$

Note que, neste caso limite, a inserção de qualquer nova aresta torna G conexo.

Portanto, se G tem $\binom{|V|-1}{2} + 1$ arestas, ele certamente será conexo.



K_{n-1}

Teorema: Seja G é um grafo conexo. G é bipartido se e somente se todo ciclo de G tem comprimento par.

Devemos provar em duas partes: a ida e a volta.

(ida) G bipartido $\rightarrow G$ contém apenas ciclos pares

1. Se G é bipartido então toda aresta é (X, Y) .
2. Se partimos 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.

(volta) G contém apenas ciclos pares $\rightarrow G$ bipartido $\equiv G$ não bipartido $\rightarrow G$ contém ciclo ímpar

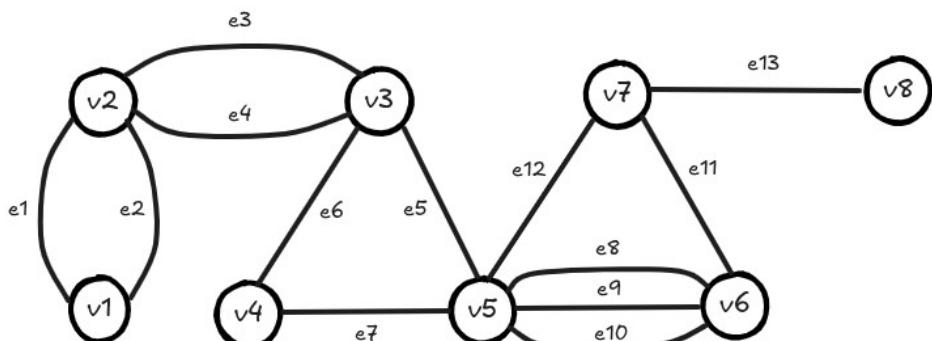
1. Se G não é bipartido, existe ao menos uma aresta (X, X) ou (Y, Y) .
2. Isso implica na existência de um triângulo em G (ciclo de comprimento 3).

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). É o maior entre os menores caminhos de G .



$$\begin{array}{llll} e(v1) = 5 & e(v2) = 4 & e(v3) = 3 & e(v4) = 3 \\ e(v5) = 3 & e(v6) = 4 & e(v7) = 4 & e(v8) = 5 \end{array}$$

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 das redes sociais (6 graus de separação)

Teorema: Seja $G = (V, E)$. Então, $r(G) \leq d(G) \leq 2r(G)$

Não iremos demonstrar, mas uma intuição pode ser verificada 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 reticulados

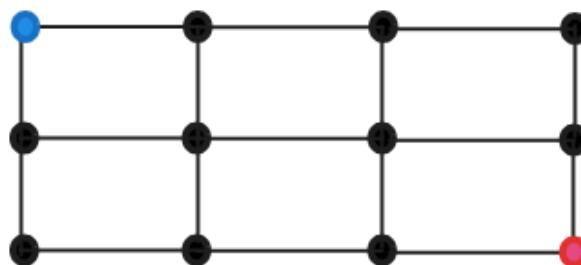
Em diversas ocasiões precisamos saber de quantas formas podemos nos locomover de um ponto a outro de um grafo. Por questões didática, vamos iniciar considerando um reticulado bidimensional pois é mais intuitivo.

Para sair do ponto azul e chegar no ponto vermelho há várias possibilidades. Alguns exemplos são

1. E, E, E, B, B
2. E, E, B, B, E
3. E, E, B, E, B

...

É relativamente simples 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?



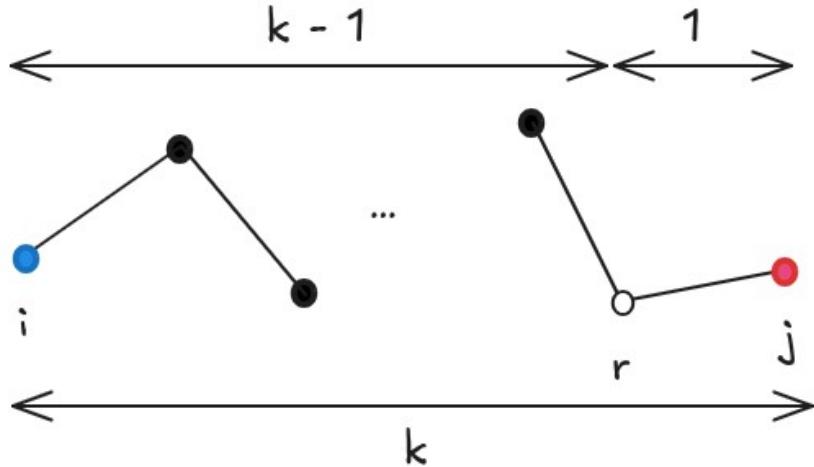
Teorema: Seja A a matriz de adjacência de G e $A^k = A \cdot A \cdot A \cdot \dots \cdot A$. O elemento $a_{ij}^{(k)}$ da matriz A^k denota o número de passeios de tamanho k que existem entre v_i e v_j

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:

$$\begin{bmatrix} & \text{linha } i \\ & a_{i1}^{(k-1)} & a_{i2}^{(k-1)} & \dots & a_{in}^{(k-1)} \end{bmatrix} \begin{bmatrix} a_{1j} & \text{coluna } j \\ a_{2j} \\ \dots \\ a_{nj} \end{bmatrix} = \begin{bmatrix} \dots & a_{ij}^{(k)} & \dots \\ \vdots & & \vdots \\ \dots & & \dots \end{bmatrix}$$

$A^{(k-1)}$ A $A^{(k)}$

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

Caminhadas aleatórias em grafos são processos estocásticos nos quais um agente percorre os vértices de um grafo escolhendo, a cada passo, um vértice vizinho de forma aleatória segundo uma distribuição de probabilidade, geralmente uniforme. Esses processos estão intimamente relacionados às **cadeias de Markov**, pois a sequência de vértices visitados forma uma cadeia de estados cuja transição depende apenas do estado atual, obedecendo à propriedade markoviana de memória limitada. Caminhadas aleatórias têm profundas implicações na teoria dos grafos, permitindo a análise de propriedades estruturais como centralidade, conectividade e comunidade, além de serem fundamentais em algoritmos para ranqueamento (como o PageRank), amostragem de grafos, espalhamento de informações em redes e modelagem de dinâmicas em sistemas complexos. A relação com cadeias de Markov também possibilita o uso de ferramentas probabilísticas e espectrais para entender o comportamento a longo prazo dessas caminhadas, como a convergência para uma distribuição estacionária.

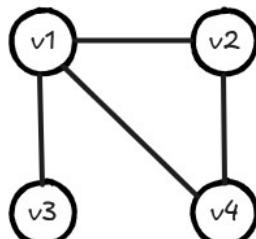
Random Walks

Uma boa analogia para entender caminhadas aleatórias em grafos é o clássico "andar do bêbado". Imagine uma pessoa embriagada em uma cidade representada por um grafo, onde cada esquina é um vértice e cada rua que liga duas esquinas é uma aresta. A cada passo, essa pessoa escolhe aleatoriamente uma rua entre as disponíveis na esquina em que está e a percorre, sem lembrar de onde veio ou planejar para onde vai, sua decisão depende apenas de sua posição atual. Essa movimentação imprevisível reflete exatamente o conceito de uma **caminhada aleatória**, onde o próximo vértice visitado é escolhido aleatoriamente entre os vizinhos do vértice atual. Assim como no andar do bêbado, esse processo pode eventualmente cobrir toda a rede ou se estabilizar em certos padrões de visitação, dependendo da estrutura do grafo.

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

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

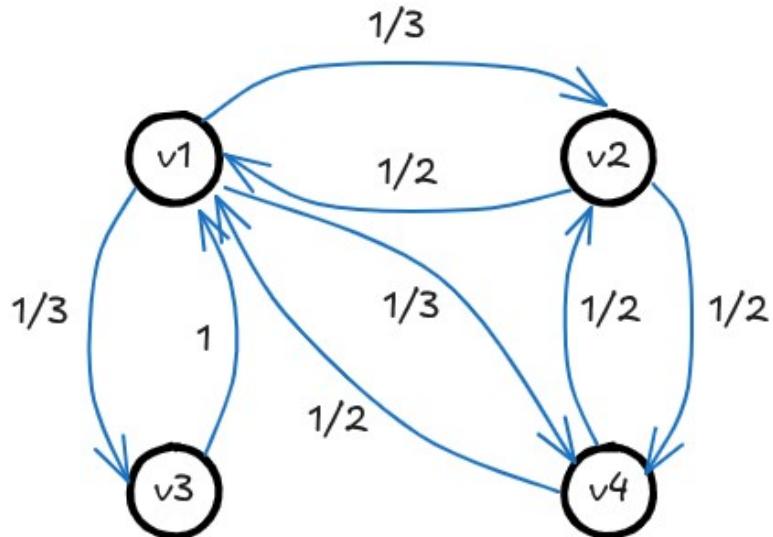
Ex: Considere o grafo a seguir:



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

Note que P não é simétrica! Além disso, cada linha de P é uma distribuição de probabilidade.

Podemos representar P como um diagrama de estados a partir de um grafo direcionado.



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

Cadeias de Markov homogêneas são processos estocásticos que evoluem em etapas discretas de tempo, nos quais a probabilidade de transição de um estado para outro depende apenas do estado atual e é **constante ao longo do tempo**. Essa propriedade de **homogeneidade** garante que a matriz de transição, que define as probabilidades de ir de um estado a outro, permanece fixa durante toda a evolução do sistema. Cadeias de Markov homogêneas são amplamente utilizadas para modelar sistemas dinâmicos com comportamento probabilístico previsível a longo prazo, como filas de atendimento, sistemas biológicos, linguagens naturais e, na teoria dos grafos, caminhadas aleatórias. Uma das principais vantagens dessa classe de cadeias é a possibilidade de analisar seu comportamento assintótico, incluindo a existência de uma **distribuição estacionária**, à qual o sistema tende após muitas iterações, independentemente do estado inicial, desde que certas condições como irreversibilidade e aperiodicidade sejam satisfeitas.

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 de todo o histórico anterior

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

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

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 passamos por 0, 1, 2, ... T - 1 só depende do último estado em que estivemos: T - 1. Por essa razão, as cadeias de Markov de primeira ordem podem ser representadas por matrizes de transição em que a probabilidade de transicionar do estado i para o estado j é dado por $P_{ij} = P(X_j | X_i)$

Em outras palavras, a probabilidade de acessar um estado s_3 no tempo t não depende do histórico todo, mas apenas de onde eu estava no tempo anterior

Equações de Chapman-Kolmogorov

As equações de Chapman-Kolmogorov são fundamentais na teoria das cadeias de Markov homogêneas, pois descrevem a relação entre as probabilidades de transição ao longo de múltiplos passos. Em essência, elas afirmam que a probabilidade de transitar de um estado i para um estado j em $n+m$ passos pode ser obtida somando-se, sobre todos os estados intermediários k, as probabilidades de ir de i a k em n passos e de k a j em m passos. Essa propriedade reflete a composição natural das transições e permite calcular probabilidades de longo prazo a partir de passos menores

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: No reino de Oz, nunca faz 2 dias nublados na sequência. Sejam os estados R, C e S para denotar rain (chuvisco), cloud (nublado) e sun (ensolarado). A matriz que governa a transição de estados em Oz é dada por:

$$P = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/2 \end{bmatrix} \quad \begin{matrix} R \\ C \\ S \end{matrix}$$

Suponha que hoje é um dia de sol, ou seja, $\vec{w}^{(0)} = [0, 0, 1]$ (começamos com certeza em um dia de sol), o que vai acontecer no longo prazo, ou seja, quem será o vetor de probabilidades $\vec{w}^{(k)}$ quando k cresce arbitrariamente, ou seja, quando $k \rightarrow \infty$? Veremos que isso 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 irredutível $\Leftrightarrow G$ 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)

Obs: Na prática para determinar se CMH é ergódica basta gerar a matriz P e computar P^k , para k suficientemente grande. Se todos os seus elementos forem não nulos, então a CMH é ergódica.

Teorema Fundamental das CM's

O teorema fundamental das cadeias de Markov estabelece as condições sob as quais uma cadeia de Markov homogênea possui uma **distribuição estacionária** e garante a **convergência** para essa distribuição ao longo do tempo, independentemente do estado inicial. Mais precisamente, o teorema afirma que, se a cadeia for **irredutível** (ou seja, é possível alcançar qualquer estado a partir de qualquer outro) e **aperiódica** (os retornos a um estado não ocorrem em múltiplos fixos de tempo), então existe uma única distribuição de probabilidade estacionária. Essa distribuição estacionária representa o comportamento estável do sistema no longo prazo: após um número suficientemente grande de passos, a probabilidade de encontrar o sistema em qualquer estado tende a essa distribuição, independentemente do ponto de partida. O teorema fundamental é central na teoria das cadeias de Markov, com implicações diretas em aplicações como análise de algoritmos estocásticos, modelagem de sistemas dinâmicos e aprendizado de máquina, especialmente em métodos como o PageRank e amostragem via Monte Carlo.

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 começa a random walk).

Power Method

O **método das potências** (ou *Power Method*) é uma técnica iterativa simples e eficiente utilizada para calcular a **distribuição estacionária** de uma cadeia de Markov homogênea, especialmente quando a matriz de transição é grande e esparsa. A ideia central do método é aproveitar o fato de que a distribuição estacionária π é um ponto fixo da matriz de transição P , satisfazendo $\pi P = \pi$. Iniciando com uma distribuição de probabilidade arbitrária $\pi(0)$, o método aplica repetidamente a operação $\pi(k+1) = \pi(k)P$, e sob certas condições — como irreducibilidade e aperiodicidade da cadeia — essa sequência converge para a distribuição estacionária única. O Power Method é amplamente utilizado em aplicações práticas como o algoritmo PageRank, onde se busca encontrar a importância relativa de páginas na web, e em simulações baseadas em cadeias de Markov, por sua simplicidade, baixo custo computacional por iteração e boa escalabilidade para grafos grandes.

Trata-se de um método iterativo para obter a distribuição estacionária de uma cadeia de Markov:

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

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

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):

$$w_i p_{i,j} = w_j p_{j,i}$$

Essa é a condição de balanço, que diz: a probabilidade de estar no estado i e transicionar de i para o estado j é igual a probabilidade de estar no estado j e transicionar de j para o estado 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:

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

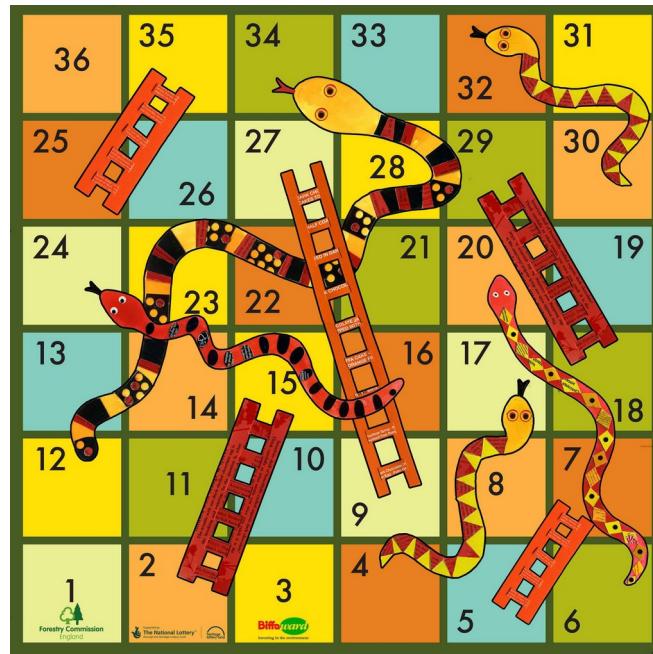
Portanto, de (*) temos que a distribuição estacionária é dada por:

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

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

O modelo **PageRank** é uma variação de caminhada aleatória em dígrafos (grafos dirigidos) amplamente conhecida por seu uso original no algoritmo de ranqueamento de páginas da web desenvolvido pelo Google. Nesse modelo, um "navegador aleatório" percorre os vértices de um dígrafo seguindo as arestas de saída com igual probabilidade, simulando o comportamento de um usuário clicando em links. Para lidar com situações em que um vértice não possui arestas de saída (nós sumidouros) ou para garantir certas propriedades matemáticas desejáveis, como a convergência para uma distribuição estacionária única, o modelo introduz um **fator de amortecimento** (tipicamente 0,85). Esse fator representa a probabilidade de seguir uma aresta real, enquanto o complemento (por exemplo, 0,15) corresponde à chance de "teletransportar-se" aleatoriamente para qualquer outro vértice do grafo. Com isso, o modelo resulta em uma cadeia de Markov irreducível e aperiódica, assegurando a existência de uma distribuição estacionária que quantifica a importância relativa de cada vértice. O PageRank é uma aplicação poderosa de métodos probabilísticos em grafos dirigidos e tem sido amplamente utilizado além da web, como em bioinformática, redes sociais e análise de citações científicas.

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

Problema: CM não é irreductível nem aperiódica, distribuição estacionária $\vec{w}^{(k)}$ não é única (depende diretamente de onde começo: $\vec{w}^{(0)}$)

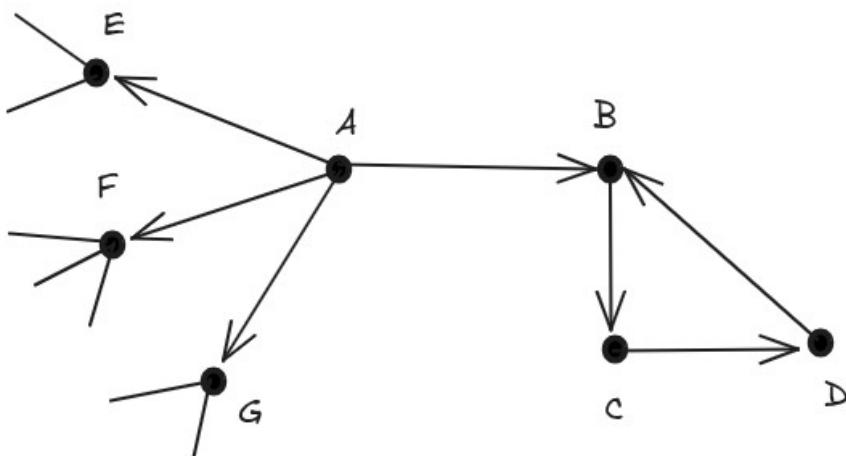
É preciso ajustar o modelo padrão para resolver problemas:

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 os demais 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 iniciamos em A no tempo $t = 0$ e vamos para B no tempo $t = 1$, caminhada aleatória torna-se periódica (BCDBCDBCD...)

Solução: permitir loops (podemos 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 (dígrafo) $\rightarrow A$ (mat. adj.) $\rightarrow P$ (probs. RW padrão) $\rightarrow \bar{P}$ (escolhido α)

Observações:

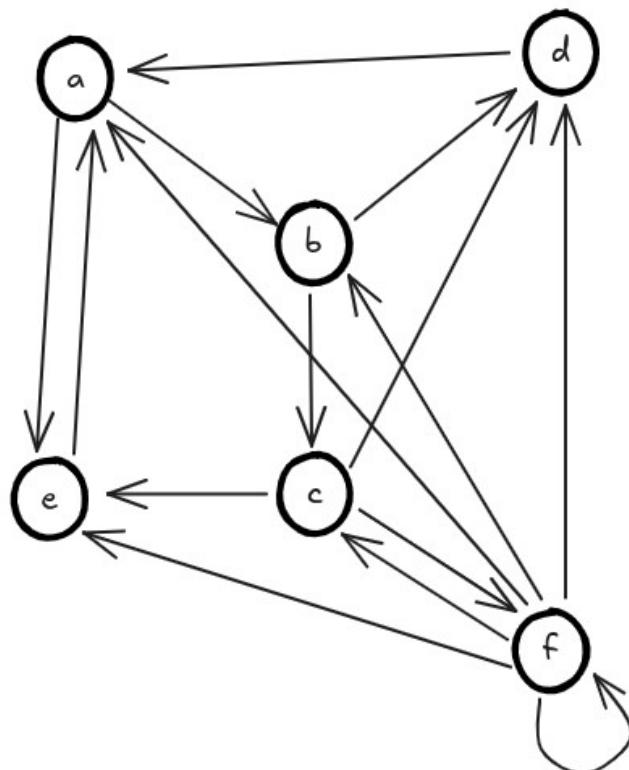
1. $\alpha=0 \rightarrow \bar{P}=P$ (RW padrão)

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 $\frac{1}{n}$

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, como era de se esperar.

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 pagerank é único. Seria um problema para o Google por exemplo se os rankings das páginas 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?

Note que 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:

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

$\frac{\vec{u}}{n}$ denota a 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

onde α denota a 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.

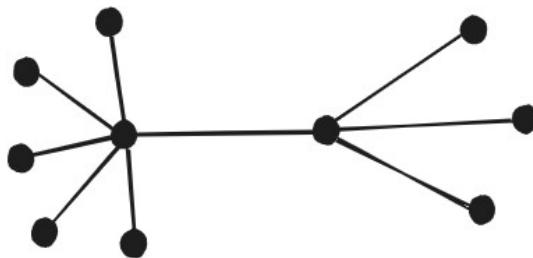
Convergência: quanto rápido \vec{w} se aproxima da distribuição estacionária?

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 estruturas fundamentais na teoria dos grafos, caracterizadas por serem grafos conexos e acíclicos. Sua simplicidade estrutural contrasta com a profundidade de suas aplicações, tornando-as indispensáveis em diversas áreas da ciência da computação, como estruturas de dados, algoritmos, redes de comunicação, compressão de dados e inteligência artificial. Em uma árvore, existe exatamente um caminho entre qualquer par de vértices, o que garante eficiência na busca e na propagação de informações. Além disso, árvores servem como base para a definição de subestruturas cruciais, como **árvores geradoras mínimas**, que permitem reduzir a complexidade de um grafo sem perder sua conectividade. Propriedades como o número fixo de arestas (sempre igual ao número de vértices menos um) e a presença obrigatória de folhas (vértices de grau um) fornecem critérios úteis para análise teórica e construção algorítmica. Por sua versatilidade e elegância, as árvores ocupam posição central no estudo e na aplicação da teoria dos grafos.

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



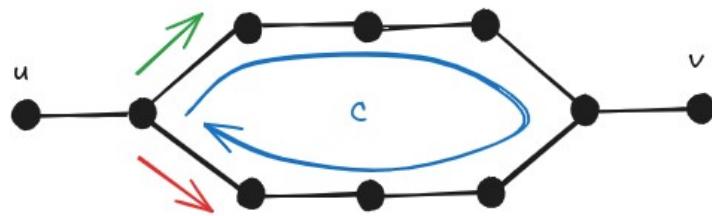
Teorema: G é uma árvore $\Leftrightarrow \exists$ um único caminho entre quaisquer 2 vértices $u, v \in V$

1. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

\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.

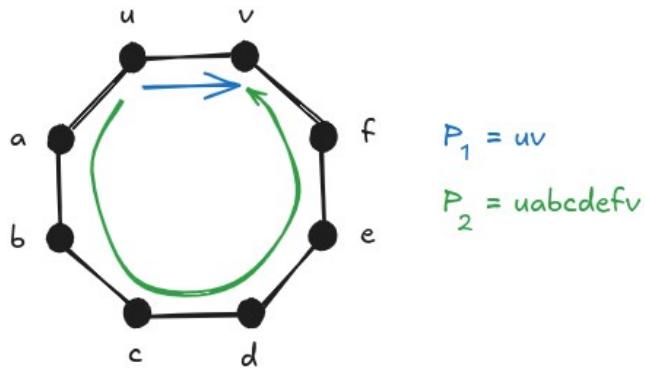


Porém neste caso temos a formação de um ciclo e portanto G não pode ser árvore.

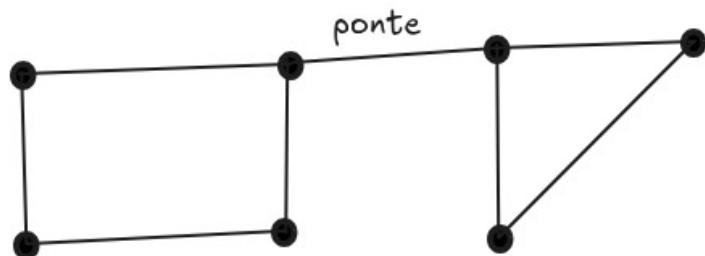
2. (volta) $q \rightarrow p = \neg p \rightarrow \neg q$

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



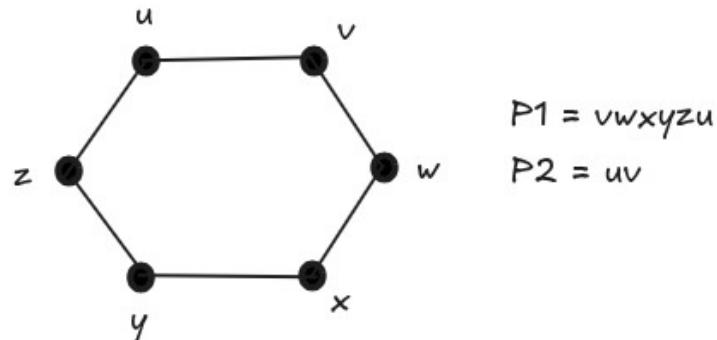
Def: Uma aresta $e \in E$ é ponte se $G - e$ é desconexo
Ou seja, a remoção de uma aresta ponte desconecta o grafo



Como identificar arestas ponte?

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

1. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$
 $e \in C \rightarrow$ aresta não é ponte



Como a 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.

2. (volta) $q \rightarrow p = \neg p \rightarrow \neg q$
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.

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

1. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$
 \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

2. (volta) $q \rightarrow p = \neg p \rightarrow \neg q$
 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.

Teorema: Seja $T = (V, E)$ uma árvore. Então, $T + e$, em que e é uma aresta que não pertence ao conjunto E , contém exatamente um único ciclo.

Prova: Como T é árvore, T é acíclico. Como T é árvore, T é conexo. Além disso, sabemos que existe um único caminho entre quaisquer dois vértices u e v . Seja e a aresta (u, v) . Haverá exatamente dois caminhos entre u e v , o que significa que temos um ciclo C . Portanto, $T + e$ tem exatamente 1 ciclo.

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

Prova por contradição:

Assumimos a negação da conclusão: Suponhamos que existe uma árvore $T = (V, E)$ com n vértices, mas que não possui $n-1$ arestas. Isso significa que ela tem um número de arestas diferente de $n - 1$.

Existem duas possibilidades para o número de arestas:

Caso 1: A árvore T tem menos de $n - 1$ arestas (ou seja, $|E| < n - 1$).

Se um grafo conexo com n vértices tem menos de $n-1$ arestas, então ele não pode ser conexo.

- **Lógica:** Para conectar n vértices, precisamos de pelo menos $n - 1$ arestas. Se tivermos menos do que isso, é impossível que todos os vértices estejam conectados entre si. Por exemplo, se tivermos n vértices e $n - 2$ arestas, o grafo terá pelo menos duas componentes conexas.
- **Contradição:** A definição de árvore inclui a propriedade de ser um grafo **conexo**. Se T tivesse menos de $n - 1$ arestas, ela não seria conexa, o que contradiz a premissa de que T é uma árvore.

Caso 2: A árvore T tem mais de $n - 1$ arestas (ou seja, $|E| > n - 1$).

Se um grafo conexo com n vértices tem mais de $n-1$ arestas, então ele deve conter um ciclo.

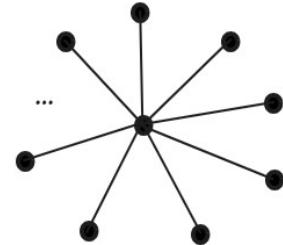
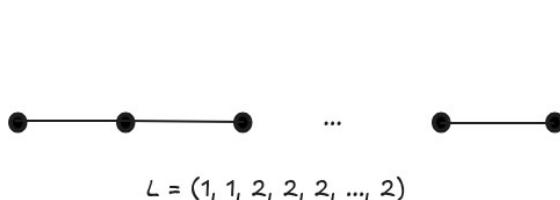
- **Lógica:** Um grafo conexo com n vértices e exatamente $n - 1$ arestas é uma árvore (e não contém ciclos). Se adicionarmos uma aresta a uma árvore, ela necessariamente criará um ciclo. Isso ocorre porque a adição de uma aresta entre dois vértices que já estão conectados por um caminho cria um novo caminho entre esses vértices, formando um ciclo com o caminho existente.
- **Contradição:** A definição de árvore inclui a propriedade de ser um grafo acíclico (sem

ciclos). Se T tivesse mais de $n - 1$ arestas, ela conteria pelo menos um ciclo, o que contradiz a premissa de T é uma árvore.

Ambas as suposições (ter menos de $n - 1$ arestas ou ter mais de $n - 1$ arestas) levam a uma contradição com a definição de uma árvore. A única possibilidade restante é que uma árvore com n vértices deve ter exatamente $n - 1$ arestas.

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

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



Teorema: Toda árvore com mais de 1 vértice possui ao menos 2 folhas (vértices de grau 1)

Prova por contradição:

Suponha que exista ao menos um vértice v tal que $d(v) = 1$

Caso a) $\nexists v \in V \mid d(v) = 1$

1. Sabemos que $|E| = n - 1$
2. Logo, $\sum_{v \in V} d(v) = 2(n-1)$ (pois T é árvore)
3. Mas, nesse caso, todos os vértices tem $d(v) \geq 2$, ou seja a soma dos graus deve ser maior ou igual a $2n$

$$\sum_{v \in V} d(v) = 2(n-1) \geq 2n \quad (\text{contradição})$$

Caso b) Existe apenas um vértice v tal que $d(v) = 1$

1. Como no caso anterior, sabemos que $\sum_{v \in V} d(v) = 2(n-1)$ (pois T é árvore)
2. Mas, como termos $(n - 1)$ vértices com $d(v) \geq 2$, a soma dos graus deve ser maior ou igual a $2(n - 1) + 1$

$$\sum_{v \in V} d(v) = 2(n-1) \geq 2(n-1)+1 \quad (\text{contradição})$$

Portanto, sempre haverá ao menos duas folhas em uma árvore.

Teorema: Toda árvore T é um grafo bipartido.

1. Escolha um vértice raiz v
2. Seja u um vértice arbitrário de T
3. Como T é árvore, existe um único caminho P_{uv}
4. Seja $f(u)$ o comprimento do caminho P_{uv} e defina

$$V_I = \{u \in V \mid f(u) \text{ é ímpar}\}$$

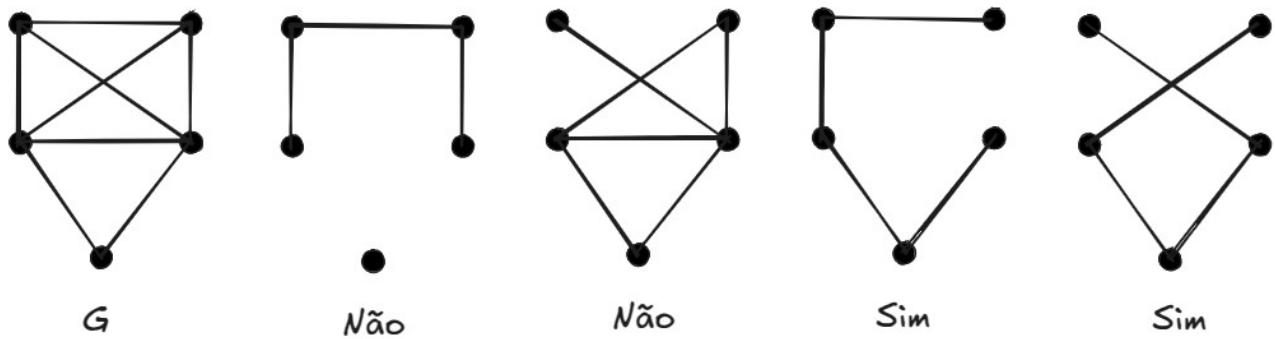
$$V_P = \{u \in V \mid f(u) \text{ é par}\}$$

5. Note que $V_I \cap V_P = \emptyset$ e $V_I \cup V_P = V$, de modo que temos uma bipartição.
6. Como T não possui ciclos, toda aresta de T tem uma extremidade em V_I e outra em V_P .
7. Portanto, T é bipartido.

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.

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)



Teorema: Seja $G = (V, E)$ um grafo conexo. Uma aresta $e \in E$ é uma ponte se e somente se ela pertence a toda árvore geradora de G .

A. (ida): $p \rightarrow q = \neg q \rightarrow \neg p$

\exists árvore geradora que não contém aresta $e \rightarrow e$ não é ponte

1. Seja T uma árvore geradora que não contém a aresta e
2. Então, T também é árvore geradora de $G - e$
3. Para quaisquer u e v arbitrários em T , existe um único caminho P_{uv} em T
4. Mas esse caminho P_{uv} também existe em $G - e$
5. Logo, $G - e$ é conexo.
6. Portanto, e não pode ser ponte (sua remoção não desconecta o grafo)

B. (volta): $q \rightarrow p = \neg p \rightarrow \neg q$ (prova por contrapositiva)

e não é ponte $\rightarrow \exists$ árvore geradora T que não contém e

1. Suponha uma aresta e não ponte
2. Então, $G - e$ é conexo e existe T que é árvore geradora de $G - e$
3. Como o conjunto de vértices de G é igual ao conjunto de vértices de $G - e$, T é árvore geradora

de G também (note que T não contém e)

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?

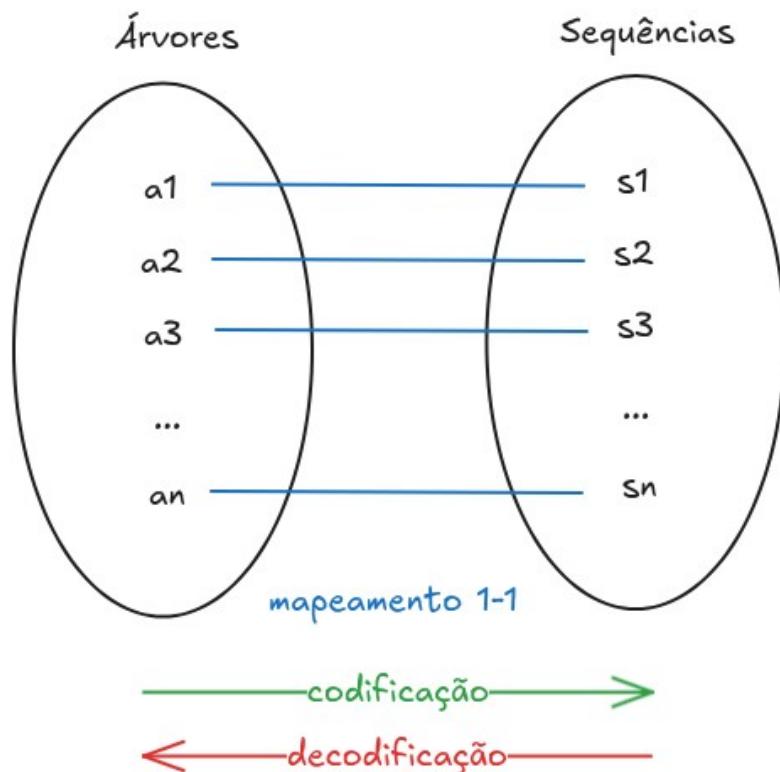
Para entender a resposta dessa pergunta iremos discutir o código de Prüfer.

Código de Prüfer

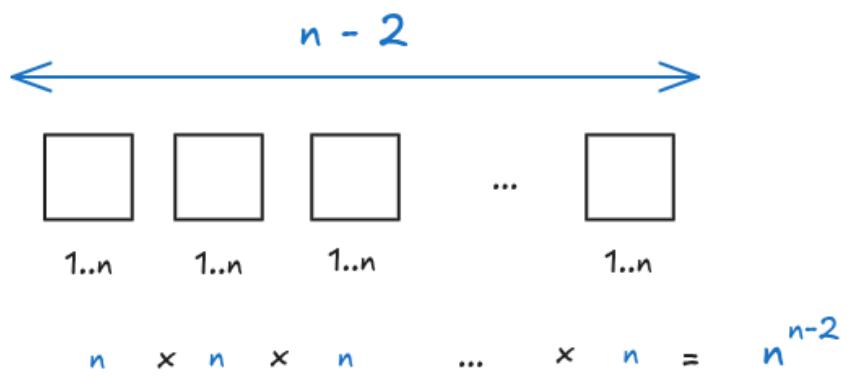
O Código de Prüfer é uma sequência única de $(n-2)$ inteiros que representa uma árvore rotulada com n vértices. Desenvolvido por Heinz Prüfer em 1918, ele oferece uma forma elegante e biunívoca de codificar e decodificar árvores, estabelecendo uma conexão direta entre o número de árvores rotuladas em n vértices e as permutações, o que levou à prova do Teorema de Cayley. Essa codificação é particularmente útil em combinatória e teoria dos grafos, permitindo a enumeração e a geração sistemática de árvores, além de ser um excelente exemplo da beleza da correspondência entre diferentes estruturas matemáticas.

Dada uma rotulação dos vértices, é um código que identifica unicamente cada possível árvore com $n > 2$ vértices. De maneira bem grosseira, é como se fosse o CPF de uma árvore, cada possível árvore distinta tem o seu próprio 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.

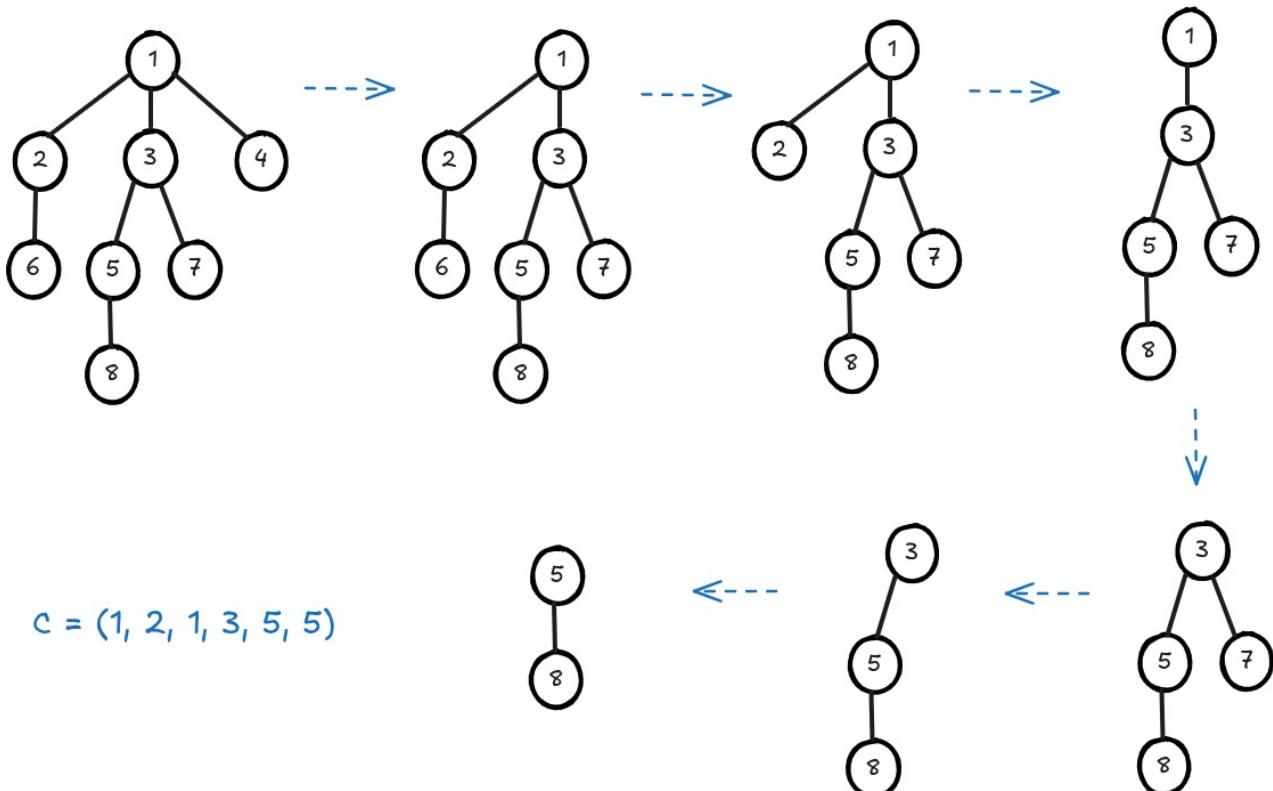
Ou seja, isso significa que o número de árvores de n vértices é n^{n-2}

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.



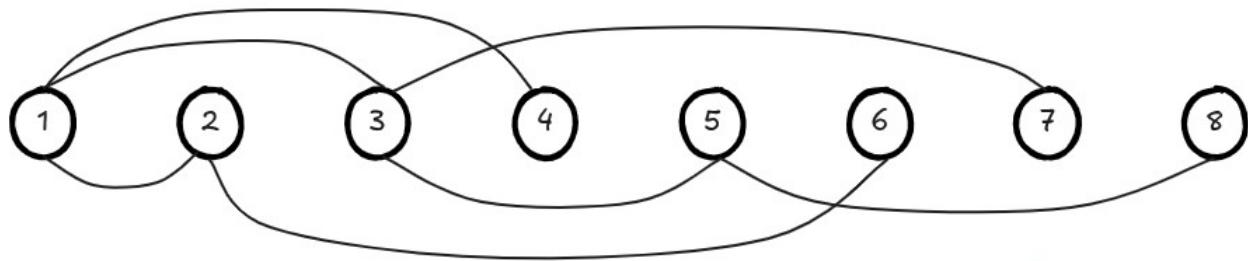
Algoritmo: Decodificação de Prüfer

Entrada: Código C

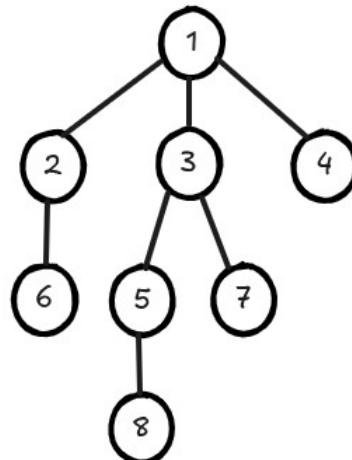
1. Representar a árvore inicial como o grafo nulo de n vértices
2. Definir o conjunto $S = \{1, 2, 3, \dots, n\}$
3. Buscar em S o menor inteiro que não está no código C. Chamaremos esse número de s_{\min}
4. Adicionar à árvore T a aresta formada pelo elemento mais a esquerda de C e s_{\min}
5. Excluir 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

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

$$S = (1, 2, 3, 4, 5, 6, 7, 8)$$



1. $e = (4, 1)$
2. $e = (6, 2)$
3. $e = (2, 1)$
4. $e = (1, 3)$
5. $e = (7, 3)$
6. $e = (3, 5)$
7. $e = (5, 8)$



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.

O problema da árvore geradora mínima (MST)

Dentre todas as árvores geradoras de G , obter aquela de menor peso.

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

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

Por exemplo, deseja-se conectar os bairros da cidade com fibra ótica. Qual é a interligação que minimiza o custo?

Estratégia gulosa: abordagem iterativa em que a cada passo devemos escolher a aresta de menor custo que seja segura (uma aresta é segura se, ao ser adicionada em T , T continua sendo uma árvore)

A seguir apresentamos uma função genérica para o problema da árvore geradora mínima.

```
Generic_MST(G, w) {
    T = ∅
    while T não for uma árvore geradora {
        encontre uma aresta segura (u, v)
        T = T ∪ {(u, v)}
    }
    return T
}
```

A pergunta que surge é: como podemos determinar se uma aresta é segura? Veremos a seguir dois algoritmos que utilizam critérios diferentes para definir o que são arestas seguras: os algoritmos de Kruskal e Prim.

O algoritmo de Kruskal

Objetivo: a cada passo escolha a aresta (u, v) de menor custo que não forme um ciclo em $T + (u, v)$

Ideia: iniciar adicionando cada vértice de G como raiz de uma árvore e a cada passo adicionar a aresta de menor custo com extremidades em árvores distintas.

Para isso, iremos utilizar 3 primitivas básicas:

1. **Make_Set(v):** cria uma árvore contendo um único vértice v (raiz)

```

Make_Set(v) {
    v.p = v      # pai do vértice v é ele mesmo (raiz)
    v.rank = 0    # altura da árvore (nível)
}

```

2. Find_Set(v): retorna qual é a árvore que o vértice v pertence (com path compression)

```

Find_Set(v) {
    if v != v.p          # se não é raiz
        v.p = Find_Set(v.p)  # recursão: v se torna o pai
    return v.p            # retorna raiz da árvore de v
}

```

3. Union(u, v): faz a fusão das raízes das árvores de u e de v, criando uma única árvore

```

Union(u, v) {
    Link(Find_Set(u), Find_Set(v))
}

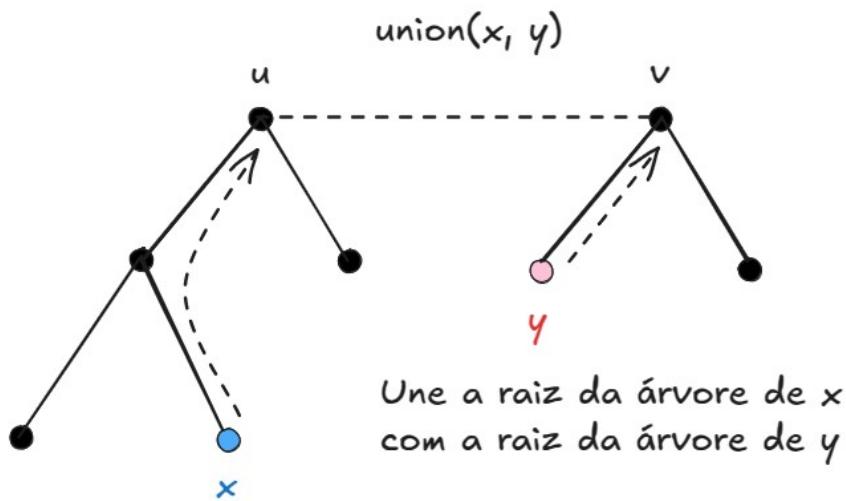
```

```

# Função auxiliar para fundir as árvores de u e de v
Link(u, v) {
    if u.rank > v.rank
        v.p = u
    else {
        u.p = v
        if u.rank == v.rank
            v.rank = v.rank + 1
    }
}

```

A figura a seguir ilustra o processo.



Ao realizar $\text{Union}(x, y)$, primeiro encontramos as raízes das árvores de x e de y , que nesse caso são u e v respectivamente. Então, a função auxiliar Link , realiza a fusão dessas duas árvores, criando uma única. Porém, a pergunta é: quem será pai de quem? Note que nesse caso u será pai de v , pois a altura da árvore de raiz u é maior! Assim, não preciso alterar a altura dela. A maior árvore domina a menor. Somente quando as alturas das duas árvores são iguais é que precisamos incrementar a altura (rank) em uma unidade. A seguir apresentamos o algoritmo de Kruskal para a obtenção de uma árvore geradora mínima.

```

MST_Kruskal(G, w) {
    T = ∅
    for each v ∈ V
        Make_Set(v)
    Crie uma lista de todas as arestas e ∈ E
    Ordene a lista pelos pesos w(e) em ordem crescente
    for each e = (u, v) na lista ordenada {
        if Find_Set(u) ≠ Find_Set(v) {
            T = T ∪ {(u, v)}
            Union(u, v)
        }
    }
}

```

Note que trata-se de um algoritmo guloso pois ele sempre tenta incluir a aresta de menor peso a cada iteração. A seguir iremos realizar um trace do algoritmo (simulação passo a passo). Para isso iremos considerar a seguinte notação:

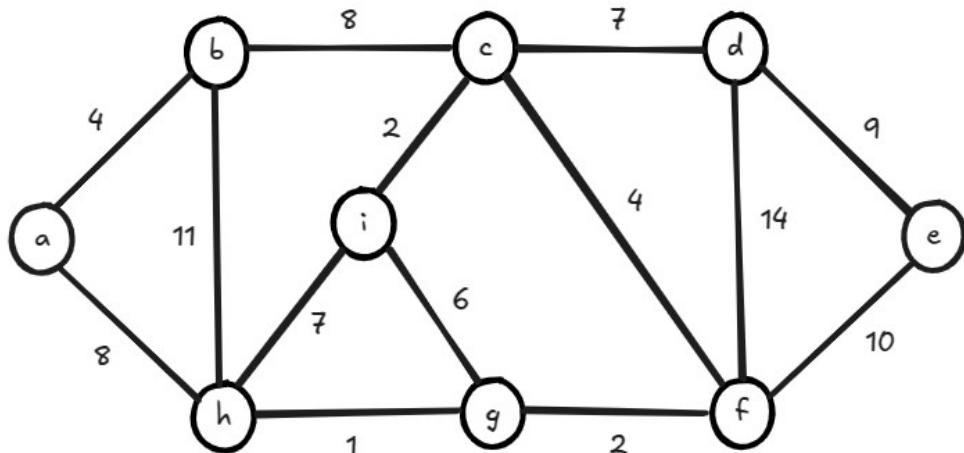
E^- : conjunto das arestas de peso mínimo não seguras ($\text{find_set}(u) = \text{find_set}(v)$)

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 com fibra ótica. Deseja saber qual é o custo mínimo de interligar todos os bairros.

Para isso, devemos encontrar a MST do grafo G, o que pode ser feito com o algoritmo de Kruskal.



A lista das arestas ordenadas por peso é:

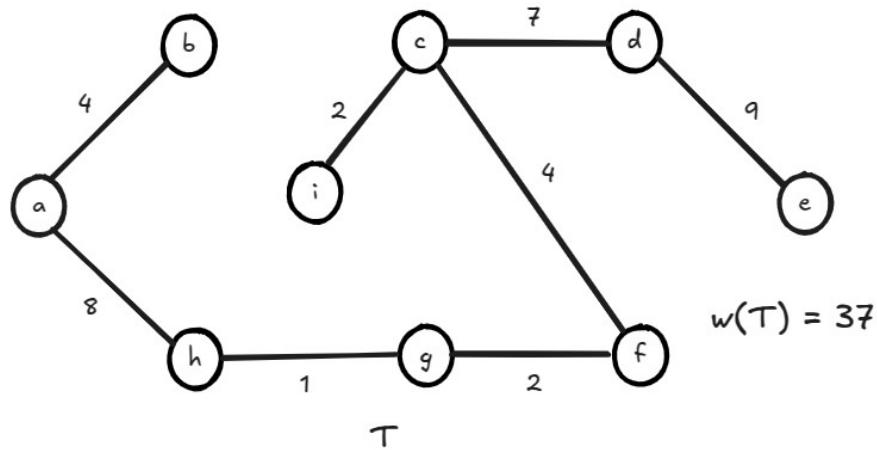
$$E_w = [1, 2, 2, 4, 4, 6, 7, 8, 8, 9, 10, 11, 14]$$

$$E = \{(g, h), (c, i), (f, g), (a, b), (c, f), (g, i), (c, d), (a, h), (b, c), (d, e), (e, f), (b, h), (d, f)\}$$

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)

A MST resultante do algoritmo de Kruskal é ilustrada a seguir.



A seguir iremos demonstrar a optimalidade do algoritmo de Kruskal, ou seja, que o algoritmo de Kruskal sempre retorna uma MST de G .

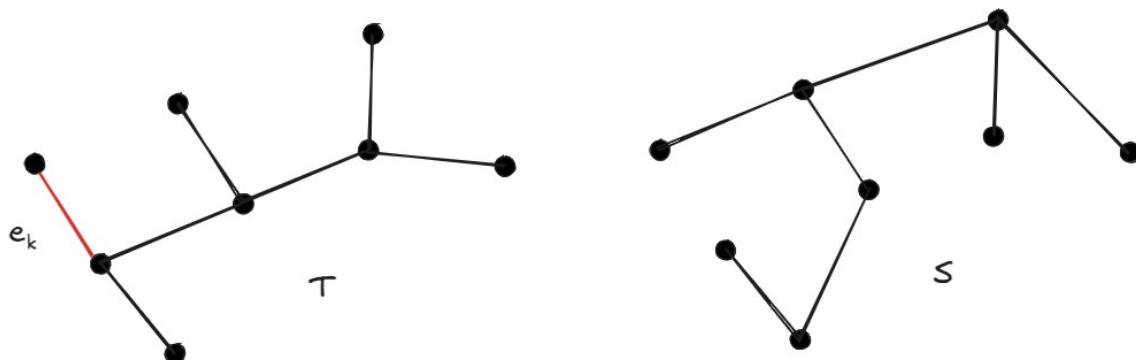
Teorema: Toda árvore T gerado pelo algoritmo de Kruskal é uma MST de G

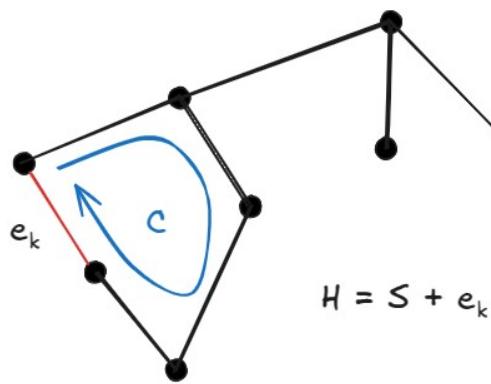
Esse resultado garante que o algoritmo sempre funciona e é ótimo (retorna sempre a solução ótima)

Prova por contradição

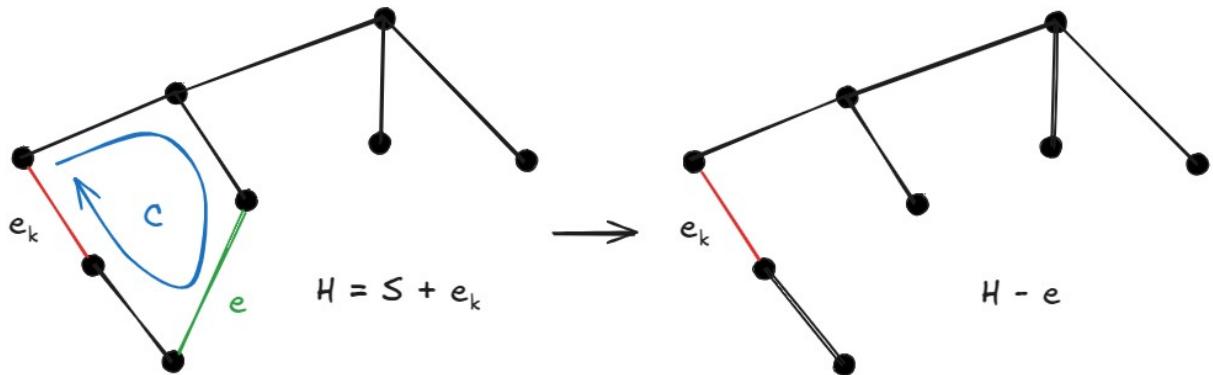
Seja T é a árvore retornada por Kruskal.

1. Suponha 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 .



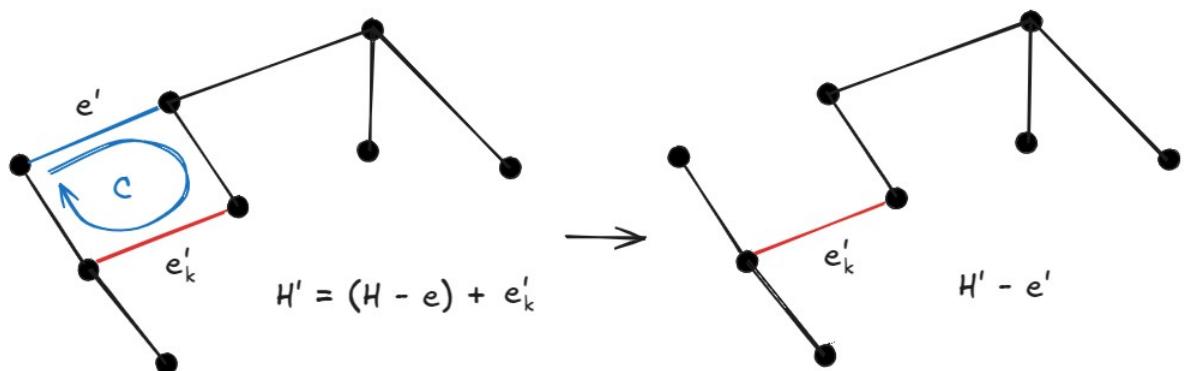
5. Porém, $w(e_k) \leq w(e)$ e assim $w(H - e) \leq w(S)$ (pois de acordo com Kruskal e_k vem antes de e na lista ordenada de arestas, é garantido pela ordenação)

Lista de arestas (após ordenação)



$$e_k \text{ precede } e \rightarrow w(e_k) \leq w(e)$$

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 .



Se continuarmos com esse processo, a árvore inicial S será transformada na árvore T, pois:

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

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).

Análise da complexidade

Primeiramente, devemos analisar as complexidades das primitivas básicas.

Pode-se perceber que a função `Make_Set(v)` é $O(1)$. Como ela é executada n vezes, temos que no total o custo será $O(n)$.

A ordenação das arestas pode ser realizada com o MergeSort ou QuickSort, que são $O(m \log m)$, onde m denota o número de arestas.

A primitiva `Find_Set(v)` deve retornar a raiz da árvore que v pertence. Note que para isso, dependemos da altura da árvore. No caso em que v é um nó folha da árvore binária, $h = \log n$. Portanto, a complexidade da função é $O(\log n)$. Se a árvore não for binária, muda-se apenas a base do logaritmo. Logo, a primitiva `Union(u, v)` que faz uso de `Find_Set(v)` também é $O(\log n)$. Note que `Find_Set(v)` é executada duas vezes para cada aresta (uma para cada extremidade: u e v). Assim, o custo total é $2m O(\log n)$, o que é $O(m \log n)$.

A primitiva `Union(u, v)` é executada somente quando uma aresta é adicionada a árvore. Como uma árvore tem $m = n - 1$ arestas, a complexidade é $O(n \log n)$. Portanto, o custo total do algoritmo de Kruskal é:

$$C = O(n) + O(m \log m) + O(m \log n) + O(n \log n)$$

Podemos observar que o termo dominante é $O(m \log m)$. Como $m < n^2$, temos que:

$$\log m < \log n^2 = 2 \log n = O(\log n)$$

ou seja, a complexidade do algoritmo de Kruskal é $O(m \log n)$.

Algoritmo de Prim

O algoritmo de Prim também utiliza uma abordagem gulosa para a construção da MST.

Ideia: iniciar de uma raiz r e a cada passo adicionar a aresta de menor custo com uma extremidade no conjunto dos vértices visitados e outra extremidade no conjunto dos vértices não visitados.

Definição das variáveis

$\lambda(v)$ ou $v.key$: menor custo de entrada para o vértice v até o momento

$\pi(v)$ ou $v.\pi$: predecessor do vértice v em T

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

Veremos a seguir que esse algoritmo é muito similar ao algoritmo de Dijkstra.

```

MST_Prim(G, w, r) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = nil
    }
    λ(r) = 0
    # Insere vértices na fila de prioridades Q
    Q = ∅
    for each v ∈ V
        Insert(Q, v)
    while Q ≠ ∅ {
        u = ExtractMin(Q)
        S = S ∪ {u}
        for each v in N(u) {
            if v ∈ Q and w(u,v) < λ(v) { # achou entrada melhor
                λ(v) = w(u,v)
                π(v) = u
                Decrease_Key(Q, v, w(u,v))
            }
        }
    }
}

```

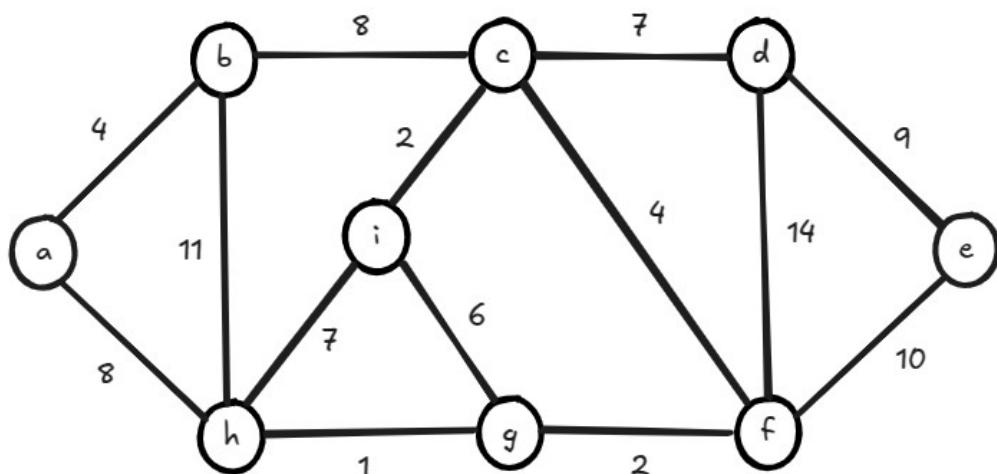
Note que o IF interno no loop FOR é equivalente a:

```

if v ∈ Q {
    λ(v) = min{λ(v), w(u,v)}
    if λ(v) was updated {
        π(v) = u
        Decrease_Key(Q, v, w(u,v))
    }
}

```

A seguir veremos um trace da execução completa do algoritmo de Prim em um simples exemplo.



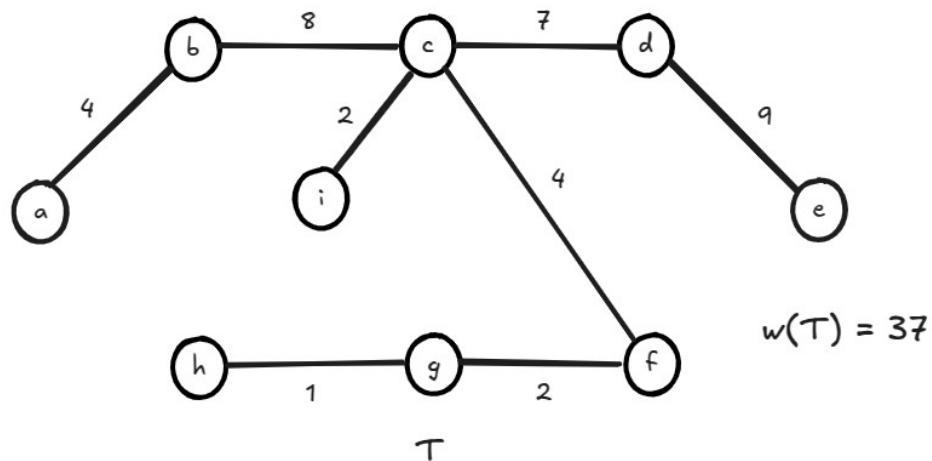
Fila de prioridades

	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	
$\lambda^{(5)}(v)$						10	2	7	
$\lambda^{(6)}(v)$						10			1
$\lambda^{(7)}(v)$									

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}	$\lambda(b) = \min\{\lambda(b), w(a, b)\} = \min\{\infty, 4\} = 4$ $\lambda(h) = \min\{\lambda(h), w(a, h)\} = \min\{\infty, 8\} = 8$	$\pi(b) = a$ $\pi(h) = a$
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}	$\lambda(h) = \min\{\lambda(h), w(i, h)\} = \min\{8, 7\} = 7$ $\lambda(g) = \min\{\lambda(g), w(i, g)\} = \min\{\infty, 6\} = 6$	$\pi(h) = i$ $\pi(g) = i$
f	{d, e, g}	$\lambda(d) = \min\{\lambda(d), w(f, d)\} = \min\{7, 14\} = 7$ $\lambda(e) = \min\{\lambda(e), w(f, e)\} = \min\{\infty, 10\} = 10$ $\lambda(g) = \min\{\lambda(g), w(f, g)\} = \min\{6, 2\} = 2$	---
g	{h}	$\lambda(h) = \min\{\lambda(h), w(g, h)\} = \min\{7, 1\} = 1$	$\pi(h) = g$
h	\emptyset	---	---
d	{e}	$\lambda(e) = \min\{\lambda(e), w(d, e)\} = \min\{10, 9\} = 9$	$\pi(e) = d$
e	\emptyset	---	---

Árvore geradora mínima



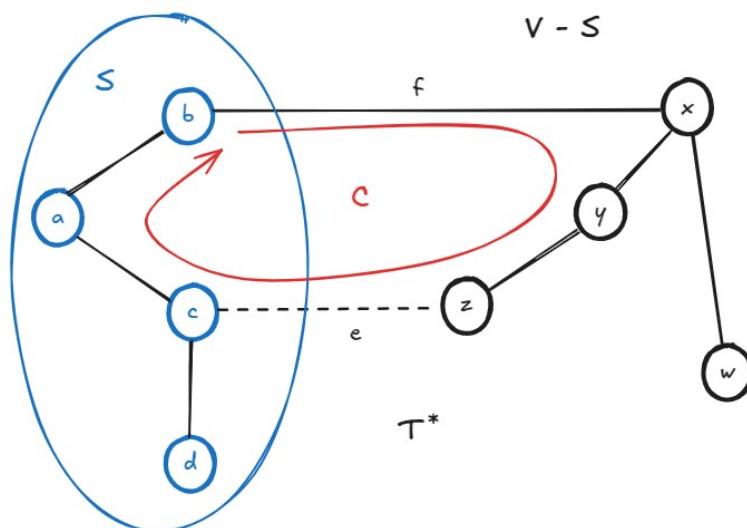
Note que, apesar da MST obtida por Prim ser diferente da MST obtida por Kruskal, ambas possuem o mesmo peso mínimo, $w(T) = 37$. Em geral, a condição para que a MST seja única é que todos os pesos das arestas do grafo $G = (V, E, w)$ sejam distintos, ou seja, não devemos ter arestas com pesos iguais no grafo de entrada.

Mapa de predecessores

v	a	b	c	d	e	f	g	h	i
$\pi(v)$	--	a	b	c	d	c	f	g	
$\lambda(v)$	0	4	8	7	9	4	2	1	2

A seguir veremos um resultado fundamental para provar a otimalidade do algoritmo de Prim.

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

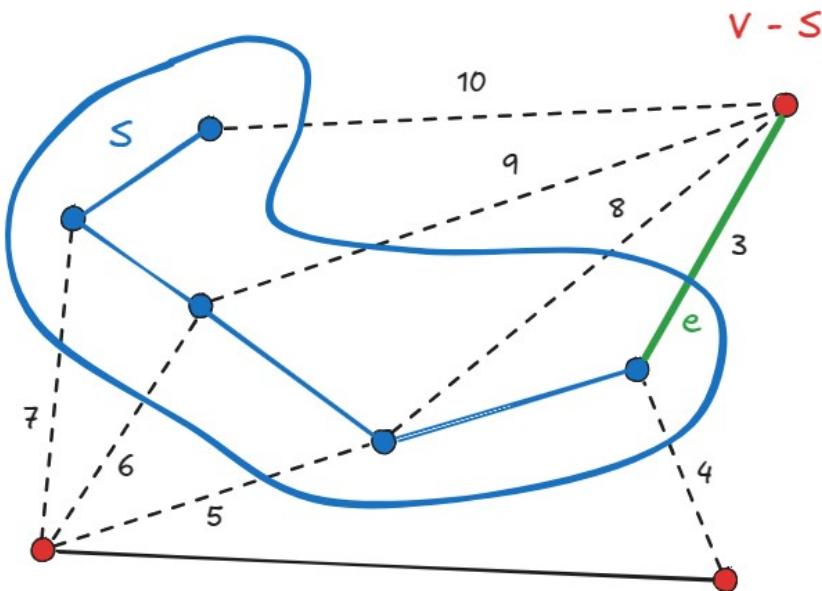


Prova por contradição:

1. Seja T^* uma MST de G .
2. Hipótese: Suponha que $e \notin T^*$ (aresta de menor peso com exatamente uma extremidade em S).
3. Ao adicionar e em T^* cria-se um único ciclo C .
4. Para que $T^* - e$ fosse uma MST, tem que haver alguma outra aresta f com apenas uma extremidade em S (senão T^* seria desconexo).
5. Então $T = T^* + e - f$ também é árvore geradora. Como, $w(e) < w(f)$, segue que T tem peso mínimo.
6. Logo, T^* não é MST de G (contradição), o que invalida a hipótese inicial de que $e \notin T^*$.

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

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



Análise da complexidade

Basicamente, há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo das estruturas de dados utilizadas.

Caso 1: $G = (V, E)$ é representado por uma matriz de adjacências
Fila de prioridades Q representada por um array simples de n elementos.

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$ (é $O(1)$ para cada vértice)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(n)$ (equivale a encontrar menor elemento do array)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n)*O(n) + (O(1) + O(1))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1)*O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: $G = (V, E)$ representado por uma lista de adjacências
Fila de prioridades Q representada por um heap binário (min-heap)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$, pois todos os pesos são iguais inicialmente (infinitos).
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = ExtractMin(Q)$ é $O(\log n)$ (dequeue), mas dentro de loop (n vezes)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (pode ter que subir no min-heap até a raiz – altura da árvore)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n \log n) + (O(1) + O(\log n))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(n \log n) + O(m) + O(m \log n)$$

Como o termo com logaritmo domina os demais: $T(n) = O((n+m) \log n)$
Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende! Devemos preferir a primeira opção se $m \log n > n^2$, o que equivale a:

$$\frac{m}{n^2} > \frac{1}{\log n} \rightarrow d > \frac{1}{\log n}$$

Em grafos mais densos, o caso 1 é mais eficiente.

Em grafos menos densos, o caso 2 é mais eficiente.

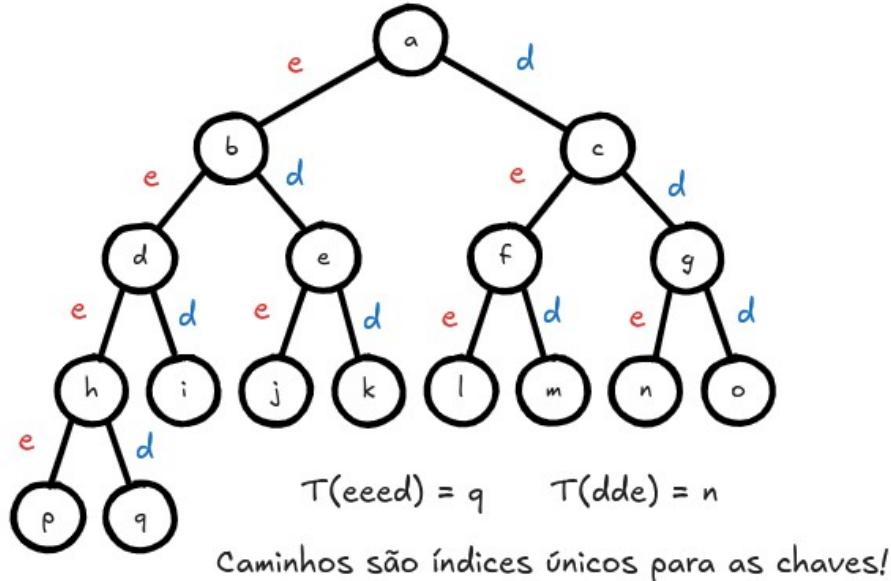
Por exemplo, se $n = 1024$, temos a seguinte regra:

- se $d > 0.1$, opção 1 é melhor
- se $d < 0.1$, opção 2 é melhor

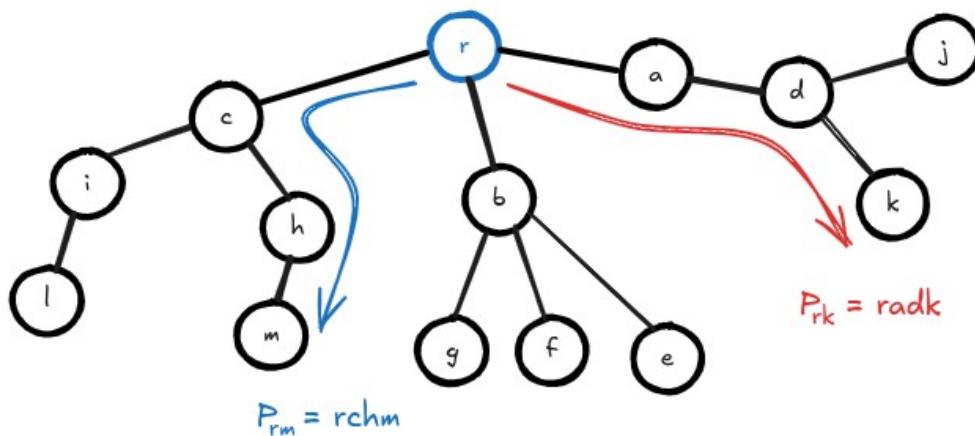
Para que d seja igual a 0.1, um grafo com $n = 1024$ vértices deve ter 104857 arestas.

Busca em grafos

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 (caminho é um índice único).



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 (princípio por trás da busca).

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 usadas no algoritmo

i) v. color : status do vértice v (existem 3 possíveis valores)

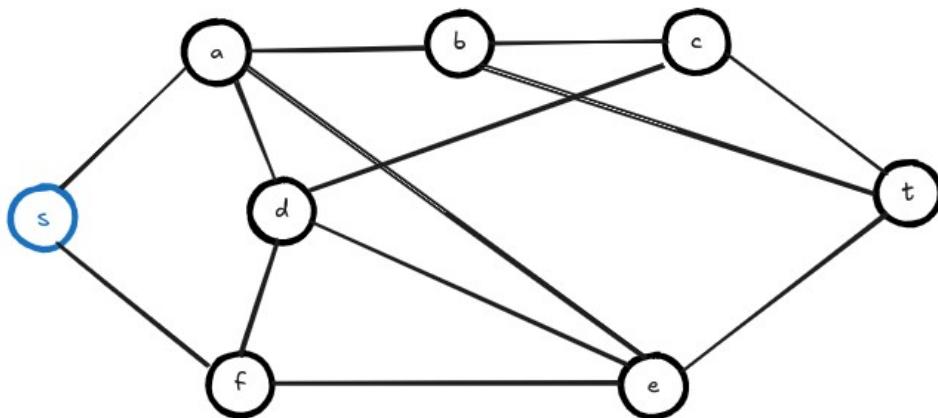
- a) WHITE: vértice v ainda não descoberto (significa que v ainda não entrou na fila Q)
- b) GRAY: vértice já descoberto (significa que v está na fila Q)
- c) BLACK: vértice finalizado (significa que v já saiu da fila Q)

- 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

ALGORITMO

```
BFS(G, s) {
    for each  $v \in V - \{s\}$  {      # inicializa as variáveis
         $v.color = WHITE$ 
         $\lambda(v) = \infty$ 
         $\pi(v) = NIL$ 
    }
     $s.color = GRAY$ 
     $\lambda(s) = 0$ 
     $Q = \emptyset$ 
    push(Q, s)           # insere raiz na fila
    while  $Q \neq \emptyset$  {          # enquanto fila não for vazia
         $u = pop(Q)$ 
        for each  $v \in N(u)$  {      # para todo vizinho de  $u$ 
            if  $v.color == WHITE$  {  # se ainda não passei aqui
                 $\lambda(v) = \lambda(u) + 1$  #  $v$  é descendente de  $u$ 
                 $\pi(v) = u$ 
                 $v.color = GRAY$ 
                push(Q, v)
            }
        }
         $u.color = BLACK$  # após processar todo vizinho, finaliza
    }
}
```

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 , $\forall v \in V$). O exemplo a seguir ilustra o trace completo do algoritmo BFS partindo do vértice s .



Ordem de acesso aos vértices

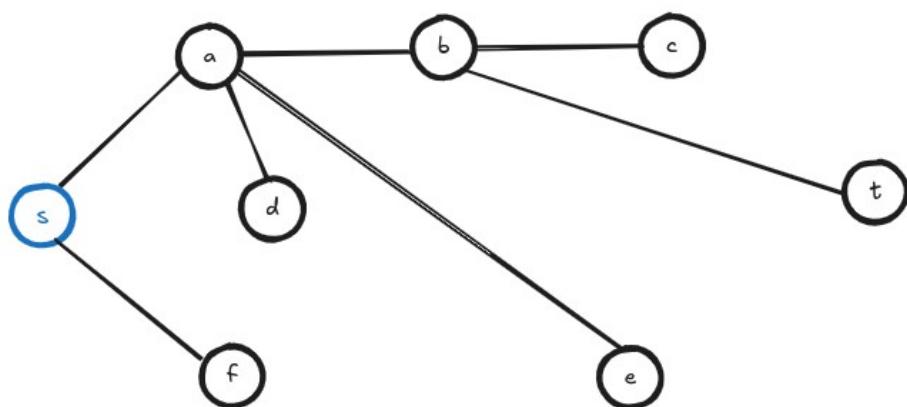
i	$u = \text{pop}(Q)$	$V' = \{v \in N(u) \mid v.\text{color} = \text{WHITE}\}$	$\lambda(v)$	$\pi(v)$
0	s	{a, f}	$\lambda(a) = \lambda(s) + 1 = 1$ $\lambda(f) = \lambda(s) + 1 = 1$	$\pi(a) = s$ $\pi(f) = s$
1	a	{b, d, e}	$\lambda(b) = \lambda(a) + 1 = 2$ $\lambda(d) = \lambda(a) + 1 = 2$ $\lambda(e) = \lambda(a) + 1 = 2$	$\pi(b) = a$ $\pi(d) = a$ $\pi(e) = a$
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

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

A complexidade da busca em largura é $O(n + m)$, onde n é o número de vértices e m é o número de arestas. É fácil perceber que cada vértice e aresta serão acessados exatamente uma vez.

Árvore BFS



Mapa de predecessores

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)

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).

Prova por contradição:

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

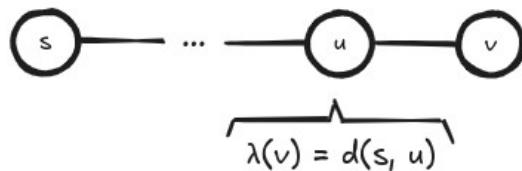
3. Então, existe caminho P_{sv} pois senão $\lambda(v) = d(s,v) = \infty$ (contradiz 2)

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



pois v foi o 1º a sair de Q com $\lambda(v) \neq d(s, v)$

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

(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 $\lambda(v) > d(s, v)$.

Busca em Profundidade (Depth-First Search - DFS)

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

Porém, 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 (overflow).

ALGORITMO (versão recursiva)

```
DFS(G, s) {
    for each u ∈ V { # inicializa as variáveis
        u.color = WHITE
        π(v) = NIL
    }
    time = 0          # variável global para contar o tempo
    for each u ∈ V {
        if u.color == WHITE
            DFS_visit(G, u)
    }
}

# Função recursiva chamada sempre que um vértice é descoberto
DFS_visit(G, u) {
    time = time + 1
    u.d = time
    u.color = GRAY
```

```

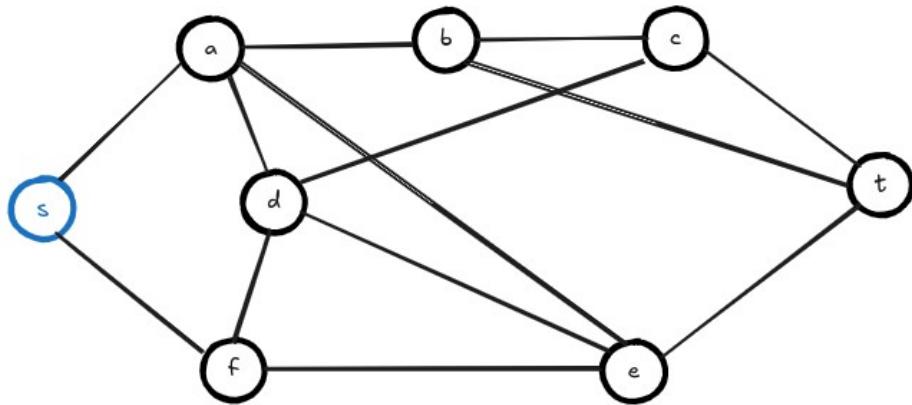
for each v ∈ N(u) {
    if v.color == WHITE {
        π(v) = u
        DFS_visit(G, v) # chamada recursiva
    }
}
time = time + 1
u.f = time
u.color = BLACK
}

```

Note que para implementar a versão iterativa (não recursiva) da Busca em Profundidade (DFS), basta usar o mesmo algoritmo da Busca em Largura (BFS) trocando a estrutura de dados fila por uma pilha.

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

O exemplo a seguir ilustra o trace completo do algoritmo DFS.



Ordem de acesso aos vértices

u	u.color	u.d	$V' = \{v \in N(u) / v.\text{color} = \text{WHITE}\}$	$\pi(v)$	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	∅	e	8
t	G	9	∅	e	10

Diferenças entre BFS e DFS

BFS

- possui aspecto espacial
- encontrar caminhos mínimos
- estrutura de dados fila

x

DFS

- possui aspecto temporal
- vértices de corte, ordenação topológica
- estrutura de dados pilha

Propriedades da árvore da busca em profundidade (DFS_tree)

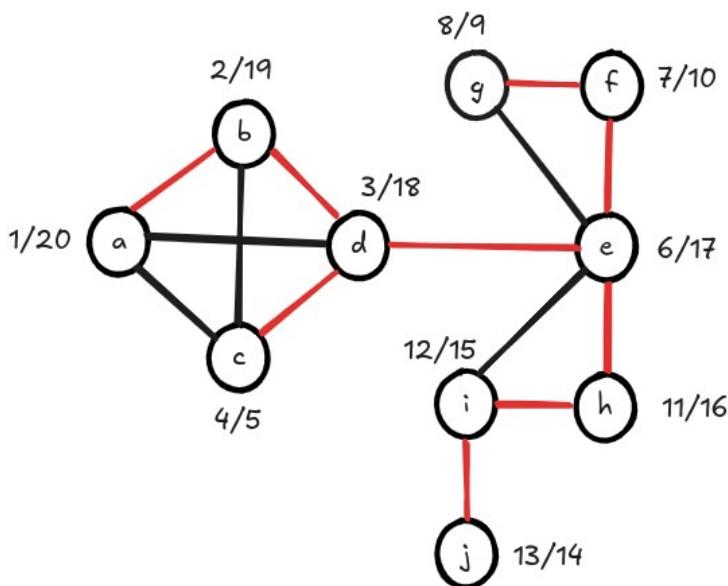
1. A rotulação tem o seguinte significado:

- 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) t_edges: $e \in T$
- b) b_edges (backward edges): $e \notin T$ (permitem voltar a um ancestral)

Def: Um vértice v é um vértice de corte em G, se e somente se v possui um filho s tal que \nexists b_edge ligando s ou qualquer descendente de s a um ancestral de v.



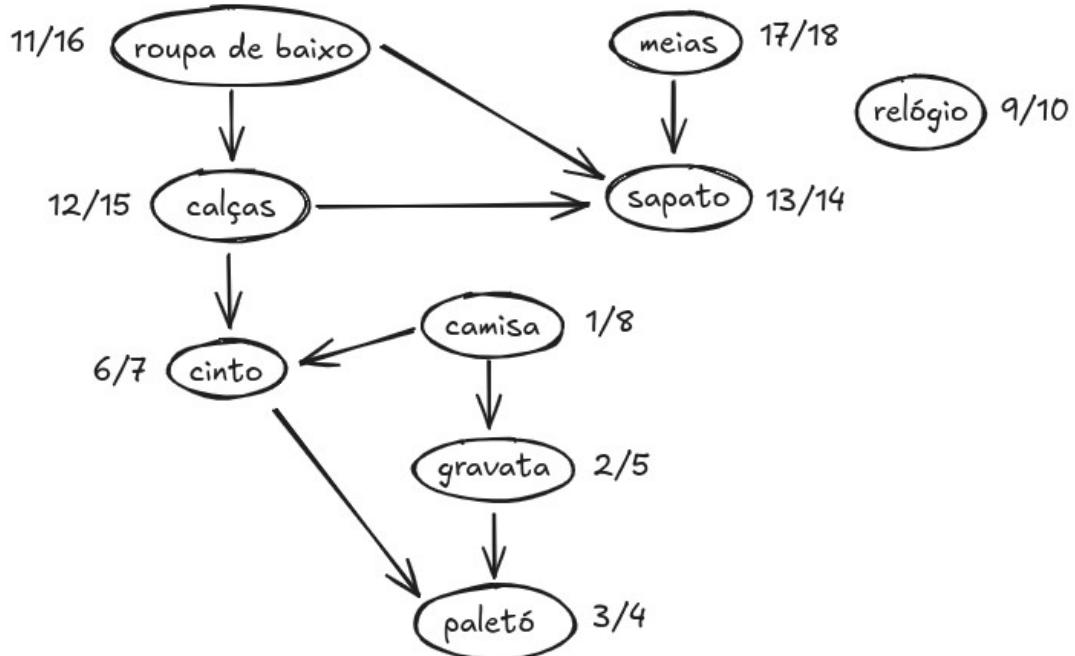
Perguntas:

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

Ordenação topológica

Uma ordenação topológica de um DAG (Directed Acyclic Graph) $G = (V, E)$ é uma ordenação linear de todos os seus vértices de modo que se G contém uma aresta (u, v) , então u aparece antes de v na ordenação. Podemos pensar na ordenação topológica de G como uma ordenação de seus vértices ao longo de uma reta horizontal de modo que todas as arestas apontam da esquerda para a direita.

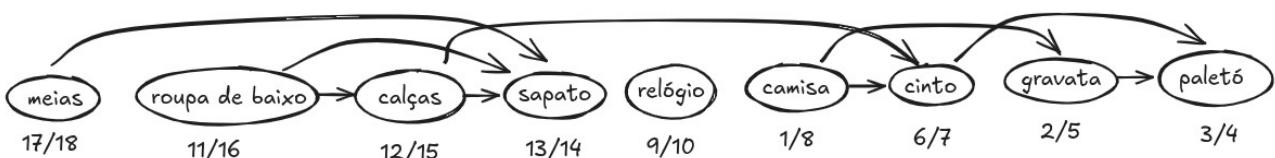
Uma aplicação da ordenação topológica seria a seguinte: imagine que você queira “ensinar” um robô humanoide a se vestir. Claramente, existe uma ordem natural que precisa ser seguida durante o processo. Por exemplo, não podemos calçar o sapato e depois colocar as meias. Suponha que o grafo acíclico direcionado que representa as dependências seja dado conforme a figura a seguir.



A rotulação obtida pela Busca em Profundidade mostra os tempos de entrada e saída para cada um dos vértices. O algoritmo Topological_Sort mostra a sequência lógica de passos necessárias para ordenar um DAG arbitrário.

```
Topological_Sort(G) {
    Execute DFS(G) para calcular v.f para todo v ∈ V
    Conforme cada vértice é finalizado, insira v.f no início de
    uma lista encadeada L
    return L
}
```

A complexidade da ordenação topológica é a mesma da Busca em Profundidade, ou seja, $O(n + m)$, onde n é o número de vértices e m é o número de arestas. O resultado da ordenação topológica do DAG anterior é ilustrada na figura a seguir.



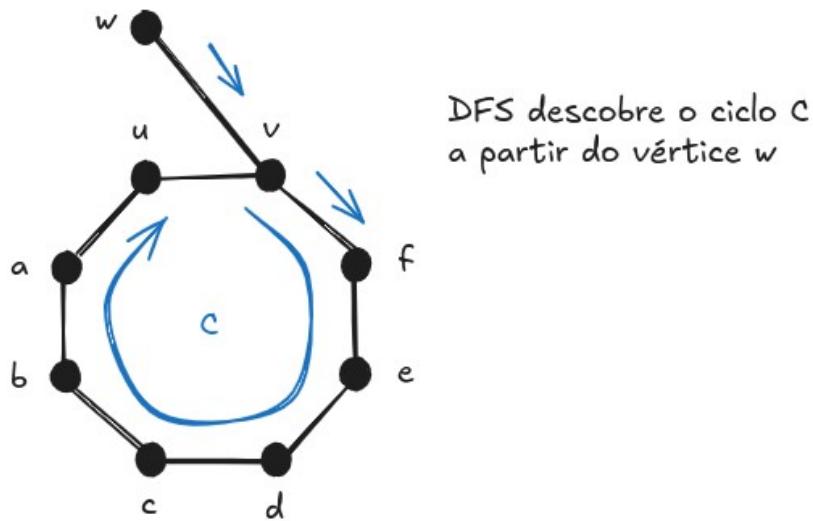
Lema: Um grafo direcionado $G = (V, E)$ é acíclico se e somente se uma busca em profundidade em G não gera b_edges.

(ida) $p \rightarrow q \equiv \neg q \rightarrow \neg p$
 G acíclico \rightarrow DFS não gera b_edge \equiv DFS gera b_edge $\rightarrow G$ possui ciclo

1. Suponha que o algoritmo DFS gere uma b_edge (u, v).
2. Então, por definição, ela liga u a um ancestral v .
3. Mas, na DFS_tree deve existir um caminho de v até u (pois é uma árvore), o que implica na geração de um ciclo.

(volta) $q \rightarrow p = \neg p \rightarrow \neg q$
 G contém ciclo \rightarrow DFS gera b_edge

1. Suponha que G contenha um ciclo C .
2. Seja v o primeiro vértice a ser descoberto em C e seja (u, v) a aresta que precede v em C .



3. Então, no tempo $v.d$, os vértices do ciclo C formam um caminho de vértices WHITE de v até u .
4. Portanto, pelas propriedades do algoritmo DFS, u será um descendente de v na DFS-tree, o que implica que (u, v) é uma b_edge (pois não irá pertencer a árvore T).

A seguir veremos um resultado que garante a corretude do algoritmo Topological_Sort.

Teorema: O algoritmo Topological_Sort realiza a ordenação topológica de um grafo direcionado acíclico G .

1. Suponha que o algoritmo DFS seja executado no DAG G para determinar $v.f$, $\forall v \in V$.
2. Basta mostrar que para qualquer par de vértices $u, v \in V$, se existe uma aresta de u para v , então $v.f < u.f$.
3. Considere uma aresta arbitrária (u, v) explorada pelo algoritmo DFS.
4. Note que quando essa aresta é explorada, v não pode ser GRAY, senão v seria um ancestral de u e a aresta (u, v) seria uma b_edge, gerando uma contradição ao lema anterior. Portanto, v deve ser WHITE ou BLACK.
5. Se v é WHITE, ele se torna um descendente de u e portanto temos $v.f < u.f$.

6. Se v é BLACK, ele já foi finalizado, de modo que o valor de $v.f$ já foi computado. Como a busca ainda está explorando u , o valor de $u.f$ ainda não foi calculado e será definido em um momento posterior. Logo, $v.f < u.f$, e a prova está concluída.

Caminhos mínimos em grafos ponderados

Encontrar caminhos mínimos em grafos é um dos mais importantes problemas da computação, em grande parte por ser utilizado em aplicações nas mais diversas áreas da ciência. Vimos que a busca em largura é capaz de encontrar caminhos mínimos em grafos não ponderados. Veremos qui como resolver o problema no caso de grafos ponderados

Def: Caminho ótimo

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

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

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

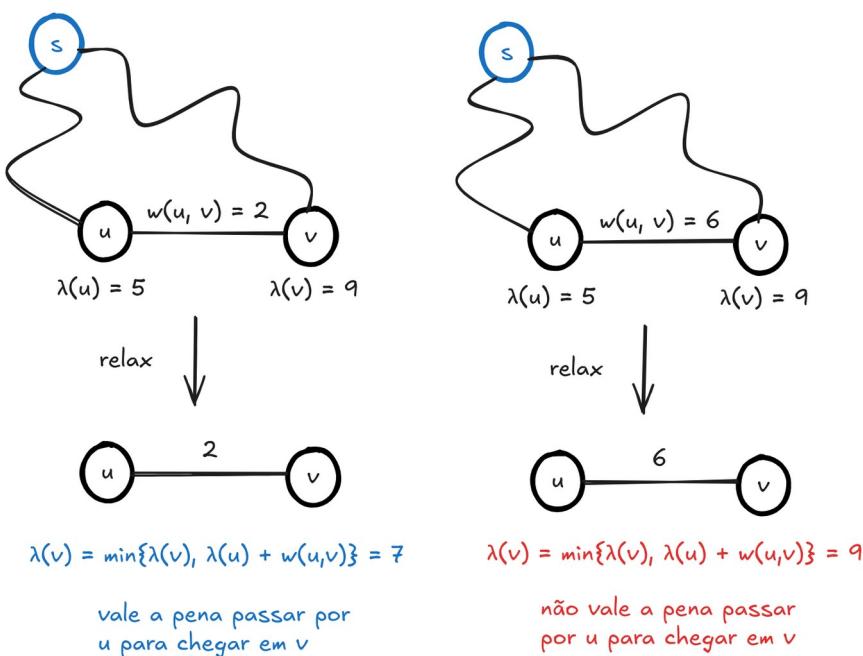
$\text{relax}(u, v, w)$: relaxar a aresta (u, v) de peso w

Para entender o que significa relaxar a aresta (u, v) de peso w , precisamos definir $\lambda(u)$ 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 ideia passar por u para chegar em v sabendo que o custo de ir de u até v é w ?



Note que a operação $\text{relax}(u, v, w)$ nunca aumenta o valor de $\lambda(v)$, apenas diminui!

ALGORITMO

```

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

```

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?

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 $\lambda(v)$)

A fila de prioridades Q possui 3 primitivas básicas:

- a) Insert(Q, v): insere um vértice v no fim da fila Q.
- b) ExtractMin(Q): remove da fila o vértice de maior prioridade.
- c) DecreaseKey(Q, v, $\lambda(v)$): modifica a prioridade do vértice v da fila Q, atualizando o seu valor de $\lambda(v)$ (note que sempre irá diminuir esse valor que é o tamanho do caminho mínimo)

S: conjunto dos vértices já finalizados (vértices para os quais o caminho mínimo já foi obtido).

```

Dijkstra(G, w, s) {
    for each  $v \in V$  {
         $\lambda(v) = \infty$ 
         $\pi(v) = \text{nil}$ 
    }
     $\lambda(s) = 0$ 
    S =  $\emptyset$ 

```

```

Q = ∅
for each v ∈ V
    Insert(Q, v)
while Q ≠ ∅ {
    u = ExtractMin(Q)
    S = S ∪ {u}
    for each v in N(u) {
        λ(v) = min{λ(v), λ(u) + w(u,v)}
        if λ(v) was updated {
            π(v) = u
            Decrease_Key(Q, v, λ(v))
        }
    }
}
}

```

Algoritmos em grafos e suas estruturas de dados

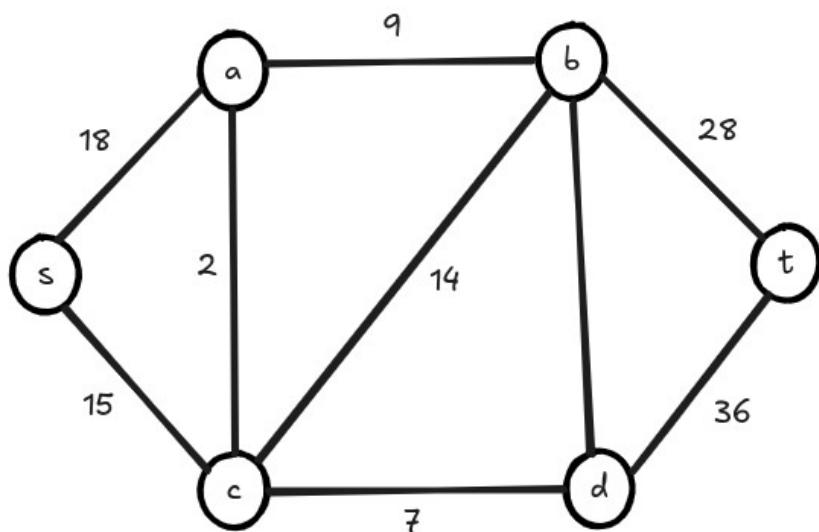
1. Busca em Largura (BFS): Fila **Q**
2. Busca em Profundidade (DFS): Pilha **S**
3. Algoritmo de Dijkstra: Fila de prioridades **Q**

Neste contexto, note que o algoritmo de Dijkstra é uma generalização da busca em largura para o caso em que os pesos das arestas não são todos iguais. Ambos crescem primeiro os menores ramos da árvore.

No algoritmo BFS toda aresta tem mesmo peso, já no algoritmo de Dijkstra esse peso é variável.

Considere o seguinte grafo ponderado. Suponha que deseja-se encontrar os menores caminhos do vértice **s** até todos os demais vértices. Para isso, basta aplicarmos o algoritmo de Dijkstra com raiz no vértice **s**.

A seguir apresentamos um trace completo (passo a passo) desse algoritmo.



Fila de prioridades

	s	a	b	c	d	t	
$\lambda^{(0)}(v)$	0	∞	∞	∞	∞	∞	
$\lambda^{(1)}(v)$		18	∞	15	∞	∞	
$\lambda^{(2)}(v)$		17	29		22	∞	
$\lambda^{(3)}(v)$			26		22	∞	
$\lambda^{(4)}(v)$			26			58	
$\lambda^{(5)}(v)$						54	

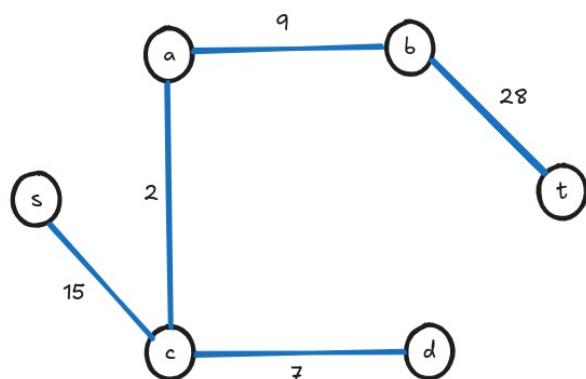
Ordem de acesso aos vértices

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}	$\lambda(c) = \min\{\lambda(c), \lambda(s) + w(s, c)\} = \min\{\infty, 15\} = 15$ $\lambda(a) = \min\{\lambda(a), \lambda(c) + w(c, a)\} = \min\{18, 17\} = 17$ $\lambda(b) = \min\{\lambda(b), \lambda(c) + w(c, b)\} = \min\{\infty, 29\} = 29$	$\pi(c) = s$ $\pi(a) = c$ $\pi(b) = c$
a	{b}	$\lambda(b) = \min\{\lambda(b), \lambda(a) + w(a, b)\} = \min\{29, 26\} = 26$	$\pi(b) = a$
d	{b, t}	$\lambda(b) = \min\{\lambda(b), \lambda(d) + w(d, b)\} = \min\{26, 32\} = 26$ $\lambda(t) = \min\{\lambda(t), \lambda(d) + w(d, t)\} = \min\{\infty, 58\} = 58$	$\pi(t) = d$
b	{t}	$\lambda(t) = \min\{\lambda(t), \lambda(b) + w(b, t)\} = \min\{58, 54\} = 54$	$\pi(t) = b$
t	\emptyset	---	---

Mapa de predecessores (árvore final)

v	s	a	b	c	d	t
$\pi(v)$	---	s	a	b	a	b
$\lambda(v)$	0	17	26	15	22	54

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



A seguir iremos demonstrar a otimalidade do algoritmo de Dijkstra.

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

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

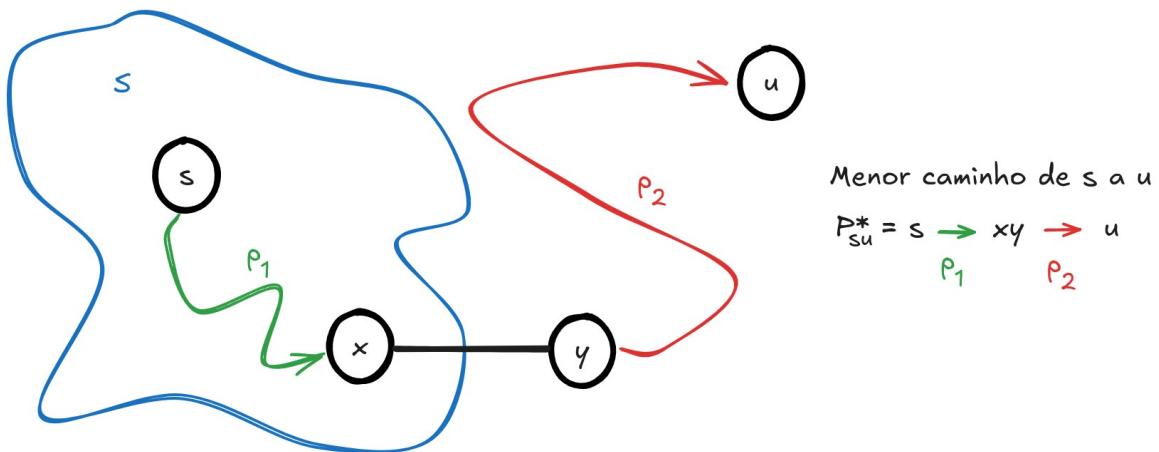
1. Suponha que u seja o 1º vértice para o qual $\lambda(u) \neq d(s, u)$ quando u entra em S .

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 necessariamente 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 $\nexists u \in V$ tal que $\lambda(u) \neq d(s, u)$ quando u entra em S .

Análise da complexidade

Há duas formas de analisar a complexidade do algoritmo de Dijkstra dependendo das estruturas de dados utilizadas.

Caso 1: $G = (V, E)$ é representado por uma matriz de adjacências

Fila de prioridades Q representada por um array simples de n elementos.

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$ (é $O(1)$ para cada vértice)
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(n)$ (equivale a encontrar menor elemento do array)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(1)$ (acesso direto)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n) + O(n)*O(n) + (O(1) + O(1))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

Sabendo que a multiplicação de dois termos lineares resulta em quadrático e que de acordo com o Hankshaking Lema, a soma dos graus de um grafo é igual a duas vezes o número de arestas, temos:

$$T(n) = O(n) + O(n^2) + O(1)*O(m)$$

Como em todo grafo básico simples $m < n^2$, temos finalmente que o algoritmo é $O(n^2)$.

Caso 2: $G = (V, E)$ representado por uma lista de adjacências

Fila de prioridades Q representada por um heap binário (min-heap)

- a) Inicialização dos $\lambda(v)$ é $O(n)$
- b) Inserção dos vértices na fila Q é $O(n)$, pois todos os pesos são iguais inicialmente (infinitos).
- c) Loop WHILE é executado n vezes (1 vez para cada $v \in Q$)
- d) $u = \text{ExtractMin}(Q)$ é $O(\log n)$ (dequeue), mas dentro de loop (n vezes)
- e) Atualização do valor de $\lambda(v)$ é $O(1)$, mas executa $k = d(u)$ vezes (loop FOR)
- f) Decrease_Key é $O(\log n)$ (pode ter que subir no min-heap até a raiz – altura da árvore)

Sendo assim a função $T(n)$ que mede a complexidade do algoritmo é:

$$T(n) = O(n) + O(n \log n) + (O(1) + O(\log n))*(d(v_1) + d(v_2) + \dots + d(v_n))$$

De modo similar ao caso anterior, podemos escrever:

$$T(n) = O(n) + O(n \log n) + O(m) + O(m \log n)$$

Como o termo com logaritmo domina os demais: $T(n) = O((n+m) \log n)$

Mas como em grafos conexos $m > n - 1$, chega-se que $T(n)$ é $O(m \log n)$.

Qual das duas implementações é mais eficiente? Depende! Devemos preferir a primeira opção se $m \log n > n^2$, o que equivale a:

$$\frac{m}{n^2} > \frac{1}{\log n} \rightarrow d > \frac{1}{\log n}$$

Em grafos mais densos, o caso 1 é mais eficiente.

Em grafos menos densos, o caso 2 é mais eficiente.

Por exemplo, se $n = 1024$, temos a seguinte regra:

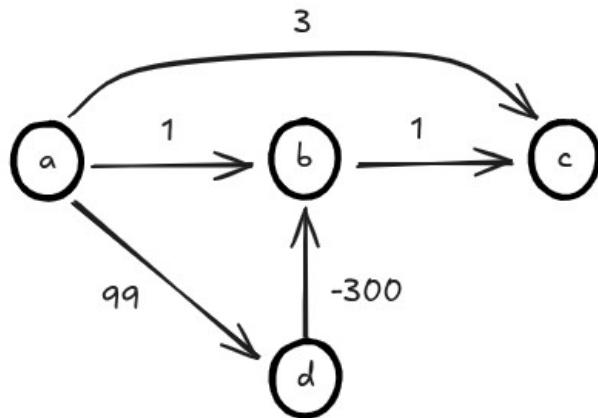
- se $d > 0.1$, opção 1 é melhor
- se $d < 0.1$, opção 2 é melhor

Para que d seja igual a 0.1, um grafo com $n = 1024$ vértices deve ter 104857 arestas.

Lembrando que o algoritmo de Djikstra tem uma limitação: em grafos com pesos negativos, sua convergência não é garantida! Ou seja, ele pode não funcionar corretamente.

Situações em que o algoritmo de Djikstra falha

O algoritmo de Djikstra pode não funcionar corretamente quando o grafo admite pesos negativos em suas arestas. Para entender o porque isso ocorre, iremos apresentar um exemplo ilustrativo. Considere o seguinte grafo ponderado. Desejamos encontrar a árvore de caminhos mínimos com raiz em A.



A seguir encontra-se a execução passo a passo do algoritmo de Djikstra.

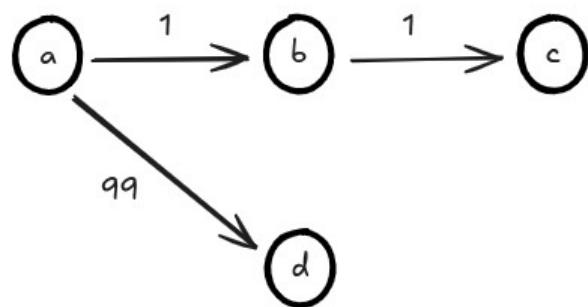
Fila de prioridades

	a	b	c	d
$\lambda^{(0)}(v)$	0	∞	∞	∞
$\lambda^{(1)}(v)$		1	3	99
$\lambda^{(2)}(v)$			1	99
$\lambda^{(3)}(v)$				99

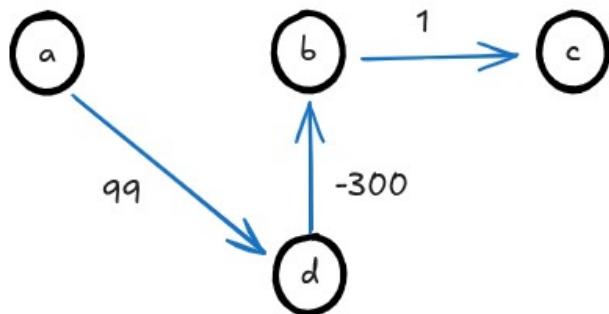
Ordem de acesso aos vértices

u	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
a	{b, c, d}	$\lambda(b) = \min\{\infty, 1\} = 1$	$\pi(b) = a$
		$\lambda(c) = \min\{\infty, 3\} = 3$	$\pi(c) = a$
		$\lambda(d) = \min\{\infty, 99\} = 99$	$\pi(d) = a$
b	{c}	$\lambda(c) = \min\{3, 1+1\} = 2$	$\pi(c) = b$
c	\emptyset	---	---
d	\emptyset	---	---

Note que ao visitar o vértice d, o vértice b já não está mais na fila. Portanto, a árvore de caminhos mínimos retornada pelo Dijkstra é a seguinte:



o que não está correto, pois deveria ser:



A pergunta natural é: como evitar que isso ocorra?

Para isso, devemos utilizar o algoritmo de Bellman-Ford, que apesar de menos eficiente do que o algoritmo de Dijkstra, é o único capaz de encontrar caminhos mínimos em grafos em que as arestas possuem custo negativo.

Algoritmo de Bellman-Ford

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

```

Bellman_Ford(G, w, s) {
    for each v ∈ V {
        λ(v) = ∞
        π(v) = NIL
    }
}
  
```

```

 $\lambda(s) = \theta$ 
for i = 1 to |V|-1 {
    for each e = (u, v) in E
        relax(u, v, w)
}

```

Para aplicar o algoritmo Bellman-Ford, o primeiro passo é definir uma ordem para que as arestas sejam relaxadas. Note que essa ordem pode ser arbitrária! Vamos considerar a seguinte ordem:

(a, b); (a, c); (a, d); (b, c); (d, b)

Como temos 4 vértices, o loop principal terá 3 iterações.

1^a iteração: (a, b): $\lambda(b) = \min\{\infty, 1\} = 1$

(a, c): $\lambda(c) = \min\{\infty, 3\} = 3$

(a, d): $\lambda(d) = \min\{\infty, 99\} = 99$

(b, c): $\lambda(c) = \min\{3, 1+1\} = 2$

(d, b): $\lambda(b) = \min\{1, 99-300\} = -201$

2^a iteração: (a, b): $\lambda(b) = \min\{1, 1\} = 1$

(a, c): $\lambda(c) = \min\{2, -201+1\} = -200$

(a, d): $\lambda(d) = \min\{99, 99\} = 99$

(b, c): $\lambda(c) = \min\{-200, -201+1\} = -200$

(d, b): $\lambda(d) = \min\{-201, 99-300\} = -201$

3^a iteração: (a, b): $\lambda(b) = \min\{1, 1\} = 1$

(a, c): $\lambda(c) = \min\{2, -201+1\} = -200$

(a, d): $\lambda(d) = \min\{99, 99\} = 99$

(b, c): $\lambda(c) = \min\{-200, -201+1\} = -200$

(d, b): $\lambda(d) = \min\{-201, 99-300\} = -201$

o que resulta na árvore desejada.

Dijkstra multisource

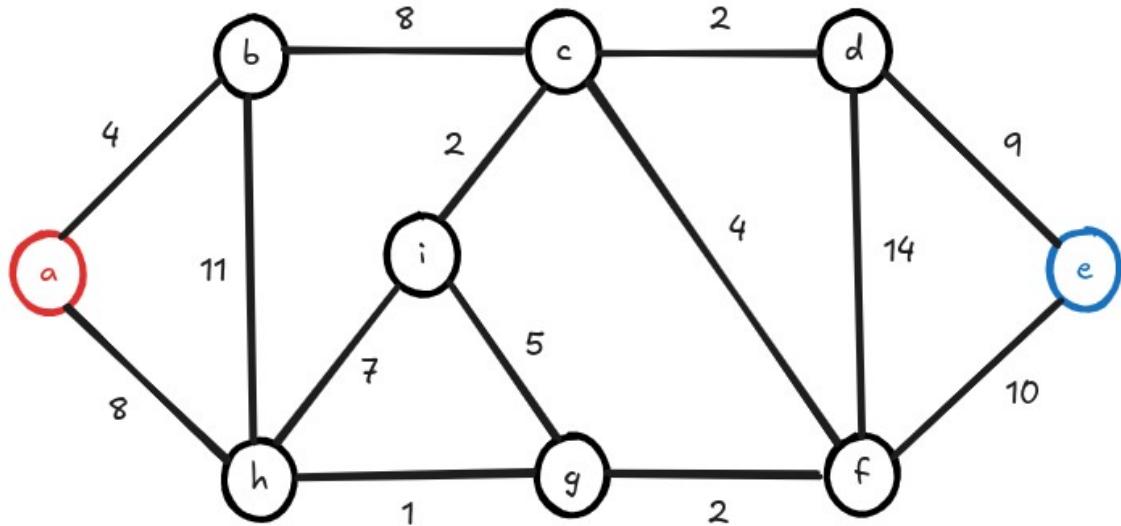
Ideia: utilizar múltiplas sementes/raízes no algoritmo de Djikstra (crescer mais de uma árvore).

Cria um 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.



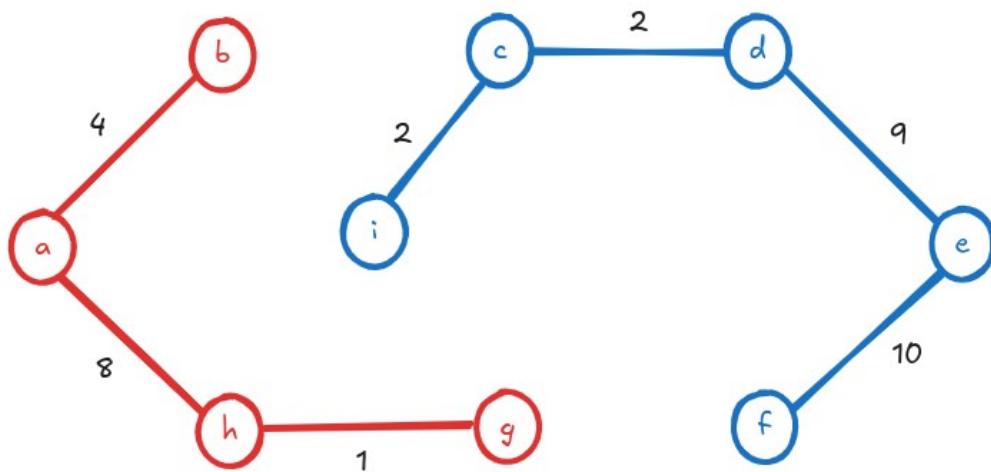
Desejamos encontrar 2 agrupamentos. Para isso, utilizaremos 2 sementes: os vértices A e E. Na prática, isso significa inicializar o algoritmo de Dijkstra com $\lambda(a)=\lambda(e)=0$

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. Um dos problemas com o algoritmo de Dijkstra é não levar em consideração nenhuma informação sobre o destino. Em grafos densamente conectados esse problema é amplificado devido ao alto número de arestas e aos muitos caminhos a serem explorados. Em suma, o algoritmo A* propõe uma heurística para dizer o quanto estamos chegando próximos do destino através da modificação da prioridades das prioridades dos vértices na fila Q. É um

algoritmo muito utilizado na IA de jogos eletrônicos.

Ideia: modificar a função que define a prioridade dos vértices

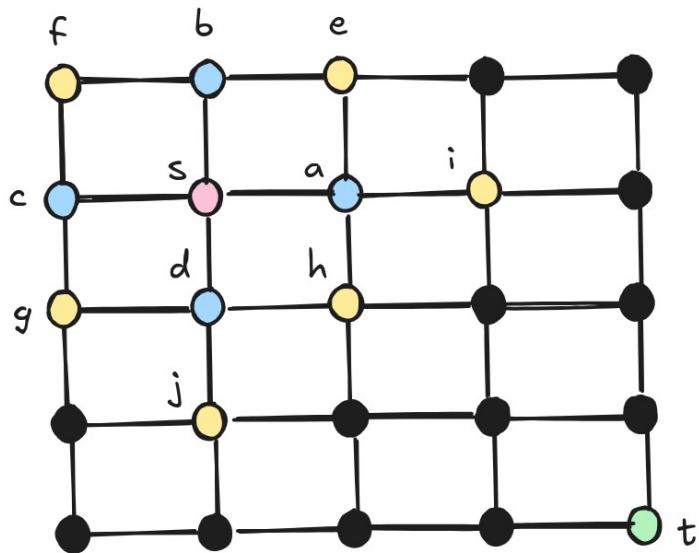
$$\alpha(v) = \lambda(v) + \gamma(v) \quad \text{onde}$$

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

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

Desafio: como calcular $\gamma(v)$?

- É viável apenas alguns casos específicos



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

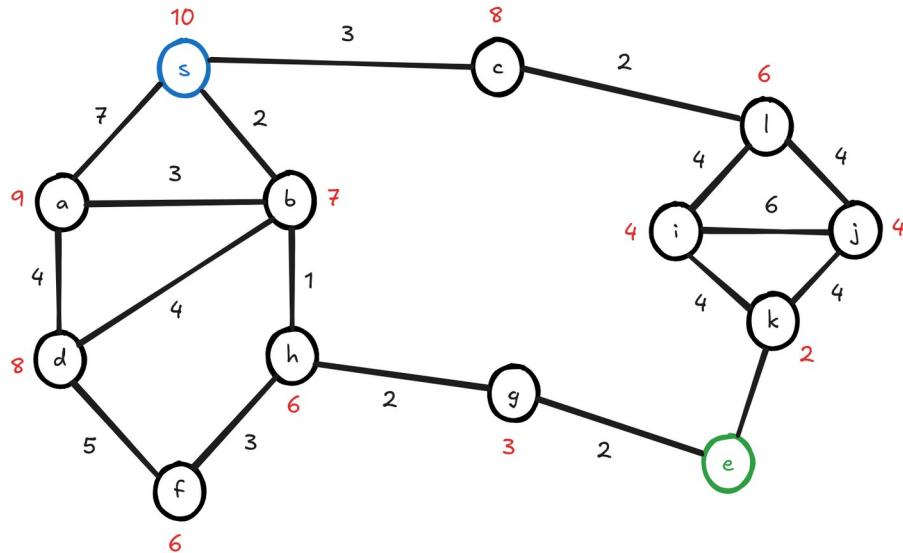
$$\lambda(e) = \lambda(f) = \lambda(g) = \lambda(h) = \lambda(i) = \lambda(j) = 2$$

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

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.

Considere o seguinte exemplo: o grafo ponderado a seguir ilustra um conjunto de cidades e os pesos

das arestas são as distâncias entre elas. Estamos situados na cidade s e deseja-se encontrar um caminho mínimo até a cidade e . O números em vermelho indicam o valor de $\gamma(v)$, ou seja, são uma estimativa para a distância de v até o destino e . Execute o algoritmo A* para obter o caminho mínimo de s a e .



Fila

	s	a	b	c	d	e	f	g	h	i	j	k	l
$\gamma^{(0)}(v)$	0	∞											
$\gamma^{(1)}(v)$		$7+9$	$2+7$	$3+8$	∞								
$\gamma^{(2)}(v)$		$5+9$		$3+8$	$6+8$	∞	∞	∞	$3+6$	∞	∞	∞	∞
$\gamma^{(3)}(v)$		$5+9$		$3+8$	$6+8$	∞	$6+6$	$5+3$					
$\gamma^{(4)}(v)$		$5+9$		$3+8$	$6+8$	7	$6+6$						

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, b, c}	$\lambda(a) = \min\{\infty, 7\} = 7$ $\lambda(b) = \min\{\infty, 2\} = 2$ $\lambda(c) = \min\{\infty, 3\} = 3$	$\pi(a) = s$ $\pi(b) = s$ $\pi(c) = s$
b	{a, d, h}	$\lambda(a) = \min\{7, 2+3\} = 5$ $\lambda(d) = \min\{\infty, 2+4\} = 6$ $\lambda(h) = \min\{\infty, 2+1\} = 3$	$\pi(a) = b$ $\pi(d) = b$ $\pi(h) = b$
h	{f, g}	$\lambda(f) = \min\{\infty, 3+3\} = 6$ $\lambda(g) = \min\{\infty, 3+2\} = 5$	$\pi(f) = h$ $\pi(g) = h$
g	{e}	$\lambda(e) = \min\{\infty, 5+2\} = 7$	$\pi(e) = g$

Note como a ordem de retirada dos vértices da fila é orientada ao destino.

Grafos Eulerianos

Grafos Eulerianos definem 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 a seguir:

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

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

(ida) G é Euleriano $\rightarrow G$ satisfaz propriedade E

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

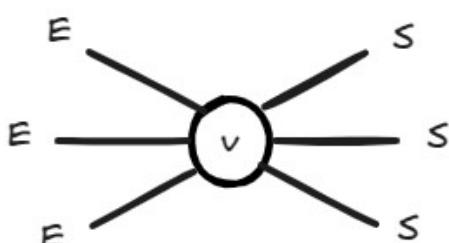
$\forall v \in V$ aparece em W (com repetições)

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

Percorra W iniciando em s .

2. Para todo $\forall v \in V$ com $v \neq s$, seja k_v o número de vezes que o vértice v é visitado.

A cada visita a v , entramos por uma aresta (x, v) e saímos por uma aresta (v, y) .



Como o tour não termina em v , segue que se consigo entrar em v , deve haver uma saída. Então, $d(v)=2k_v$, o que resulta em um número par.

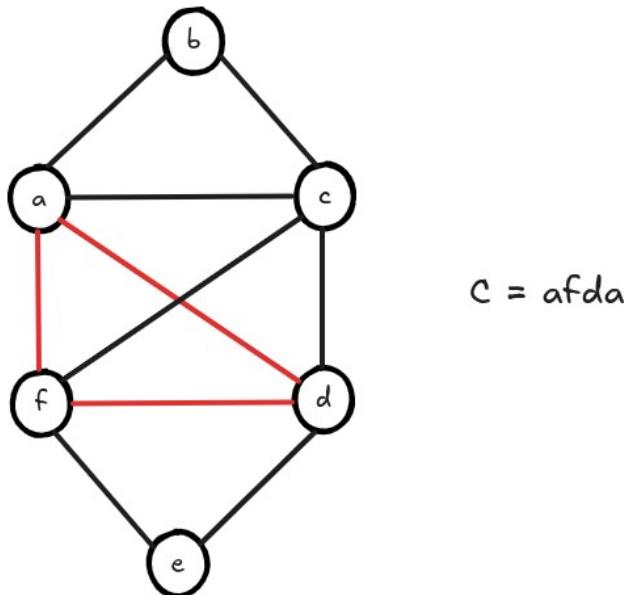
3. A única exceção é o vértice s : $d(s)=2k_s-2$, que também é par, pois na primeira visita apenas saímos de s e na última visita apenas entramos em s .

(volta) G satisfaz propriedade E $\rightarrow G$ contém um circuito Euleriano

A prova dessa afirmação é feita apresentando um algoritmo que, dado um grafo G que satisfaz E, sempre produz um tour de Euler.

1. Escolha um vértice arbitrário $v \in V$ e inicie um circuito C .

- a) A cada passo, escolha uma aresta, atravesse-a e apague-a de G .
- b) Pare apenas ao retornar a v .



2. FATO: Enquanto não retornamos a v , não podemos ficar presos (pois graus são todos pares)

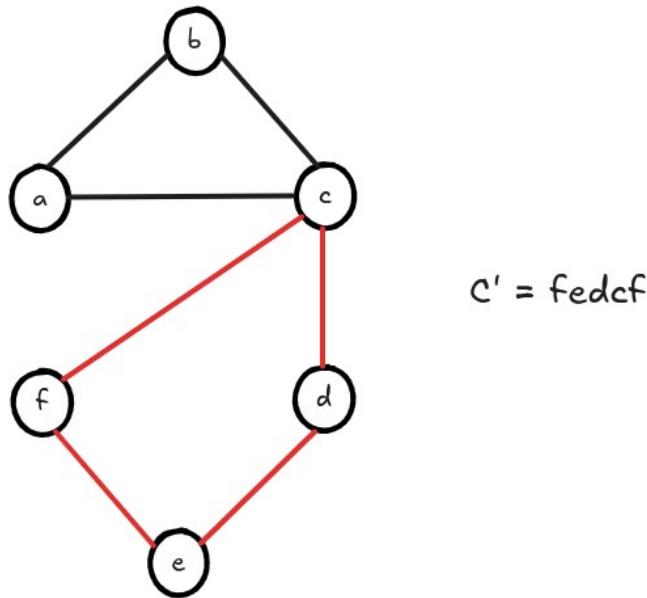
$\forall u \in V - \{v\}$ não há como ficar preso em u pois:

- Antes de cada visita a u , ele tem grau par, o que implica em
 - Enquanto visitamos u , ele tem grau ímpar, o que implica em
 - É impossível visitar u quando ele tem grau zero (não havendo saída)

3. Se o circuito C encontrado contém $\forall e \in E$, OK. Ele é Euleriano como desejado.

4. Caso contrário, um dos vértices visitados ainda possui grau positivo (> 0), pois G é conexo e ao deletar as arestas do circuito C , subtraímos 2 dos graus dos vértices $v \in C$. Isso significa que ainda há arestas a serem percorridas. Além disso, o grau de todos os vértices restante é par.

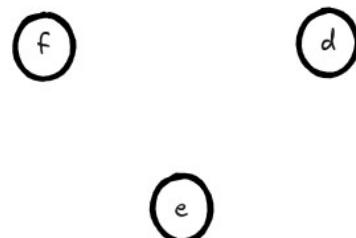
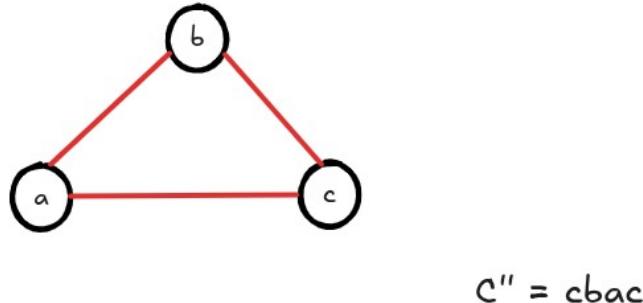
5. Escolha um vértice $v \in C$ e percorra outro circuito C , como antes, até retornar ao ponto de partida. Isso novamente é possível de ser realizado devido ao passo 2,



6. Temos agora, 2 circuitos arestas disjuntos com ao menos um vértice em comum que podem ser combinados para gerar um novo circuito C.

$$\begin{aligned} C &= a \rightarrow f \rightarrow d \rightarrow a \\ &\quad + \qquad \Rightarrow \qquad C = a \rightarrow f \rightarrow e \rightarrow d \rightarrow c \rightarrow f \rightarrow d \rightarrow a \\ C' &= f \rightarrow e \rightarrow d \rightarrow c \rightarrow f \end{aligned}$$

7. Repita os passos 3, 4, 5 e 6 até não restar mais arestas em G.



$$\begin{aligned} C &= a \rightarrow f \rightarrow e \rightarrow d \rightarrow c \rightarrow f \rightarrow d \rightarrow a \\ &\quad + \qquad \Rightarrow \qquad C = a \rightarrow f \rightarrow e \rightarrow d \rightarrow c \rightarrow b \rightarrow a \rightarrow c \rightarrow f \rightarrow d \rightarrow a \\ C'' &= c \rightarrow b \rightarrow a \rightarrow c \end{aligned}$$

8. Ao fim do processo, é obtido um circuito que contém $\forall e \in E$

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.

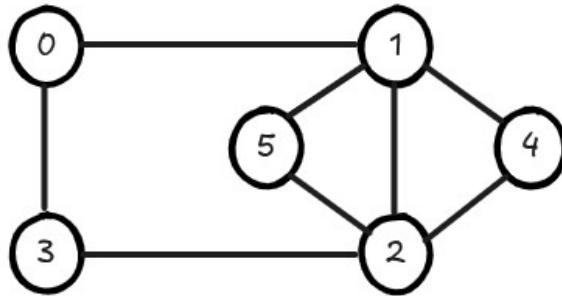
Algoritmo de Fleury

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	$[0, 1, 2, 4, 1, 5, 2, 3, 0]$

O algoritmo de Fleury é mais complicado de se implementar pois depende de quão 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

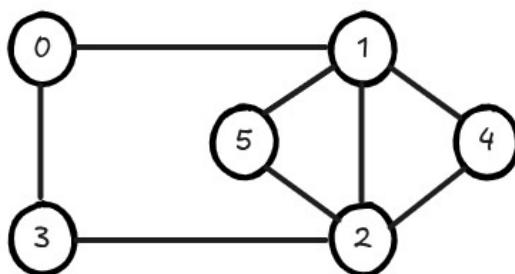
a. Seja u o vértice do topo.

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

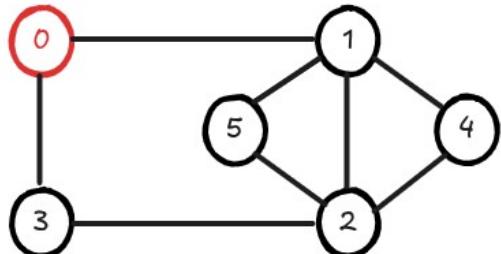
c. Se u não possui aresta incidente desmarcada, remova u do topo da pilha S e imprima u.

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

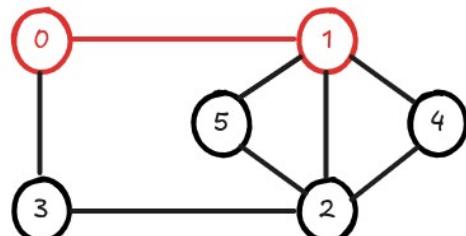
Ex:



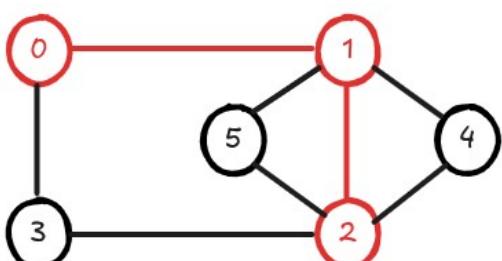
$$S = \emptyset \quad T = \emptyset$$



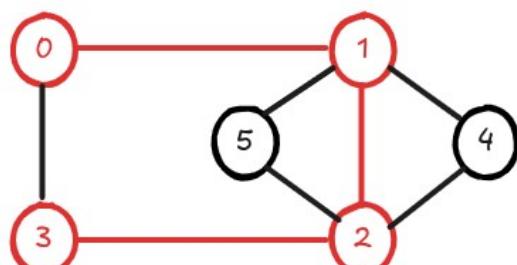
$$S = [0] \quad T = \emptyset$$



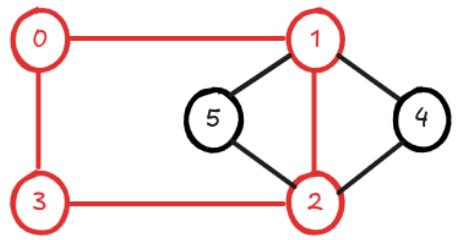
$$S = [0, 1] \quad T = \emptyset$$



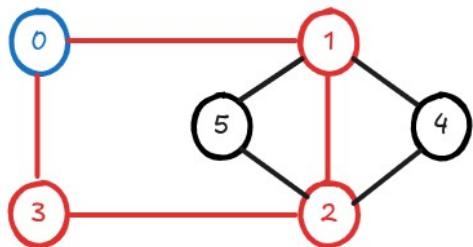
$$S = [0, 1, 2] \quad T = \emptyset$$



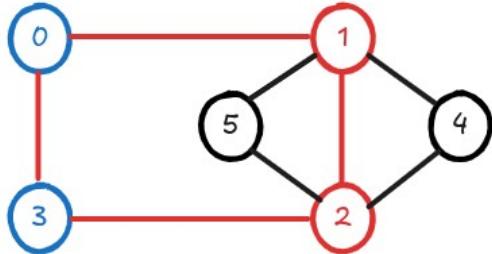
$$S = [0, 1, 2, 3] \quad T = \emptyset$$



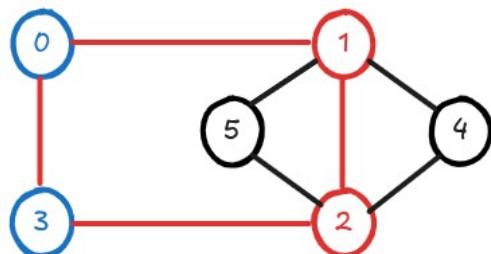
$$S = [0, 1, 2, 3, 0] \quad T = \emptyset$$



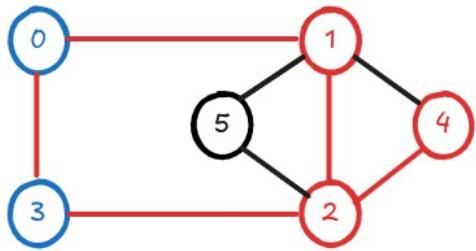
$$S = [0, 1, 2, 3] \quad T = [0]$$



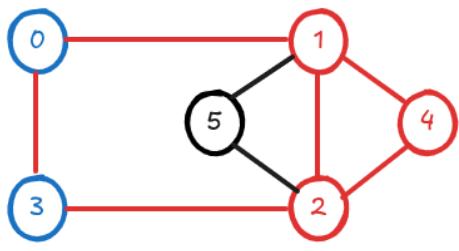
$$S = [0, 1, 2] \quad T = [0, 3]$$



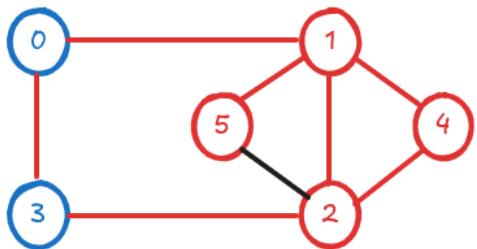
$$S = [0, 1, 2] \quad T = [0, 3]$$



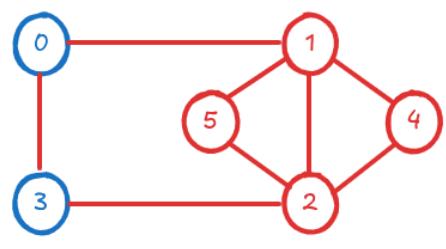
$$S = [0, 1, 2, 4] \quad T = [0, 3]$$



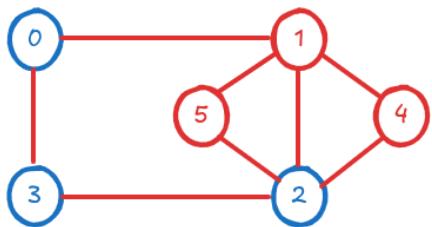
$$S = [0, 1, 2, 4, 1] \quad T = [0, 3]$$



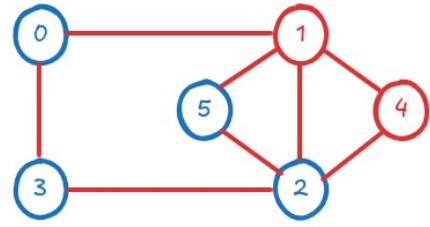
$$S = [0, 1, 2, 4, 1, 5] \quad T = [0, 3]$$



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

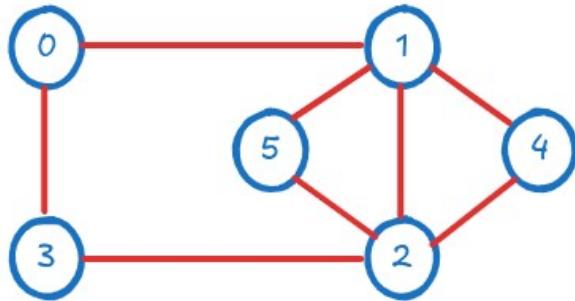


$$S = [0, 1, 2, 4, 1, 5] \quad T = [0, 3, 2]$$



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

Desempilhando os vértices de S em T, finalmente chega-se a solução final:



$$S = \emptyset$$

$$\tau = [0, 3, 2, 5, 1, 4, 2, 1, 0]$$

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: Se G é Euleriano, basta aplicar o algoritmo de Fleury ou Hierholzer.

Problema: E se o grafo G não for Euleriano?

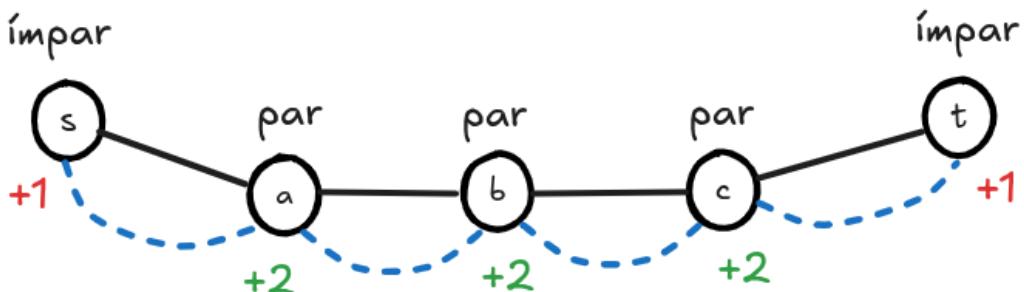
i) Transformar G num supergrafo Euleriano G^*

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

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

Obs: $E(G^*) - E(G)$ denota o conjunto das arestas novas inseridas em G

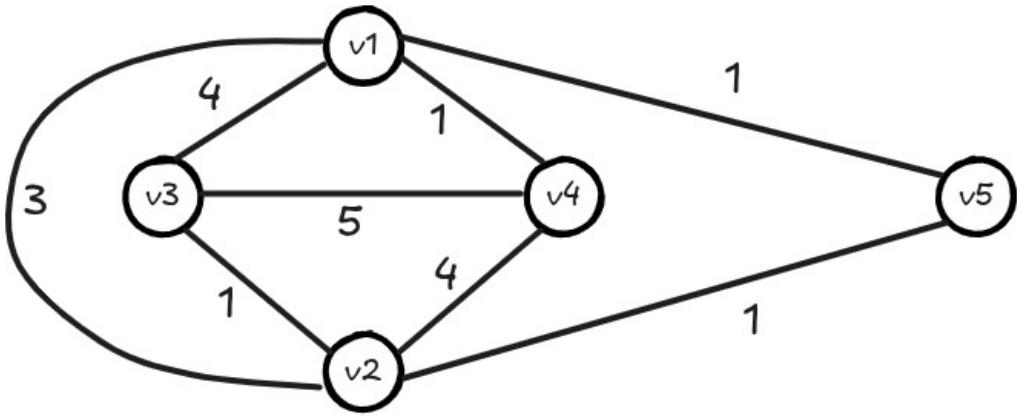
Solução: Por simplificação iremos considerar o caso mais trivial do problema: o grafo G não é Euleriano pois existem exatamente 2 vértices de grau ímpar, ou seja, $|I| = 2$.



- a) Aplicar algoritmo de Dijkstra para encontrar caminho mínimo P_{st}^* entre os vértices ímpares.
- b) Aplicar algoritmo de Fleury ou Hierholzer em G^* para obter um tour de Euleriano

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

Ex: Suponha que o vértice v_1 é o correio central. O carteiro deve sair de v_1 entregar cartas e voltar a v_1 percorrendo a menor distância possível.



1. Transformar G em G^*

Aplicar algoritmo de Dijkstra para descobrir menor caminho de v_3 a v_4

Fila

	v1	v2	v3	v4	v5	
$\lambda^{(0)}(v)$	∞	∞	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	$V' = \{v \in N(u) \wedge v \in Q\}$	$\lambda(v), \forall v \in V'$	$\pi(v)$
v3	{v1, v2, v4}	$\lambda(v_1) = \min\{\infty, 4\} = 4$ $\lambda(v_2) = \min\{\infty, 1\} = 1$ $\lambda(v_4) = \min\{\infty, 5\} = 5$	$\pi(v_1) = v_3$ $\pi(v_2) = v_3$ $\pi(v_4) = v_3$
v2	{v1, v4, v5}	$\lambda(v_1) = \min\{4, 1+3\} = 4$ $\lambda(v_4) = \min\{5, 1+4\} = 5$ $\lambda(v_5) = \min\{\infty, 1+1\} = 2$	---
v5	{v1}	$\lambda(v_1) = \min\{4, 2+1\} = 3$	$\pi(v_5) = v_2$
v1	{v4}	$\lambda(v_4) = \min\{5, 3+1\} = 4$	$\pi(v_1) = v_5$
v4	\emptyset	---	$\pi(v_4) = v_1$

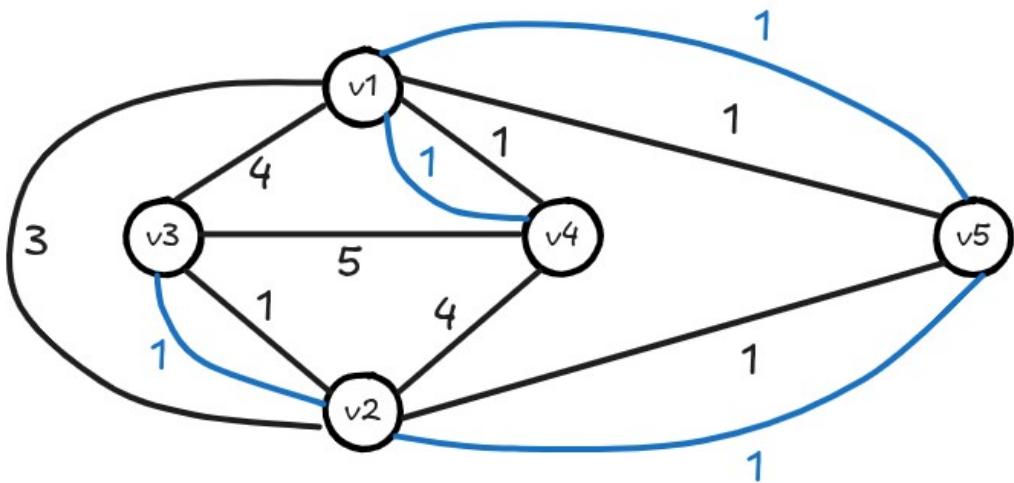
Mapa de predecessores

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

Portanto, pelo mapa de predecessores, o caminho mínimo entre v_3 e v_4 é dado por:

$$P^* = v_3 \ v_2 \ v_5 \ v_1 \ v_4$$

Assim, o supergrafo G^* fica:



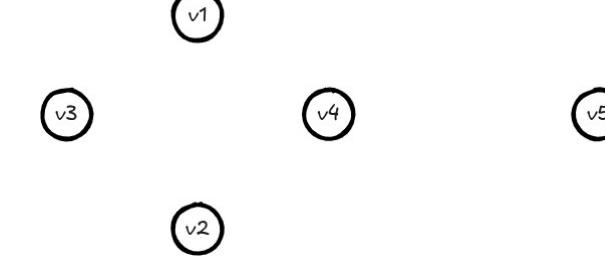
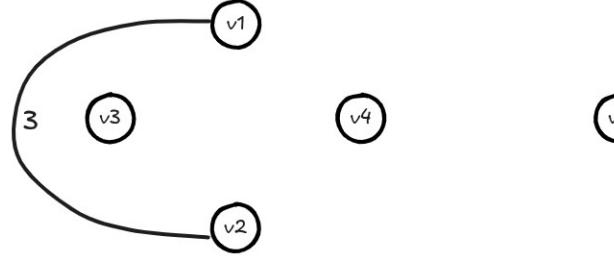
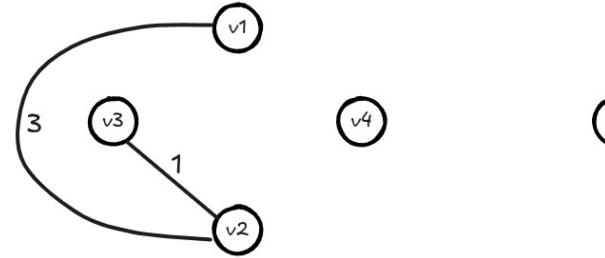
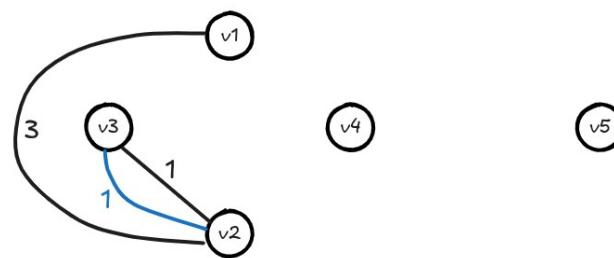
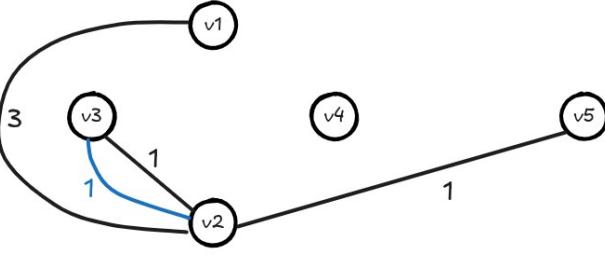
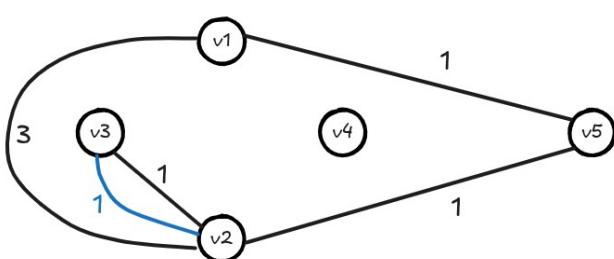
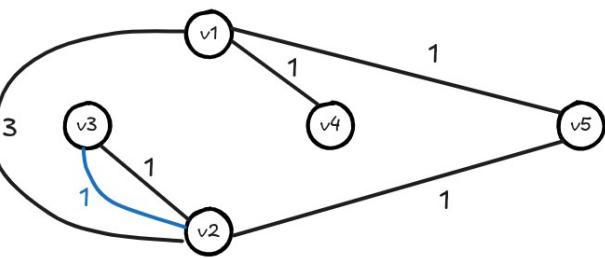
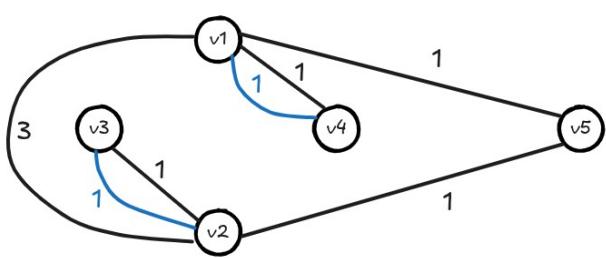
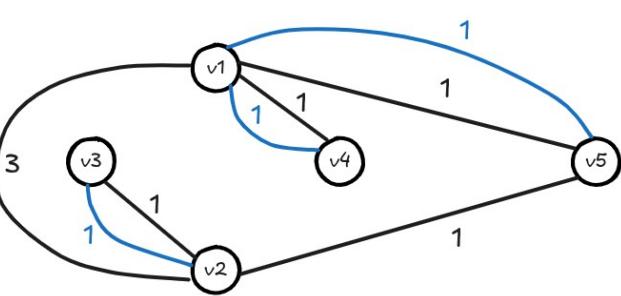
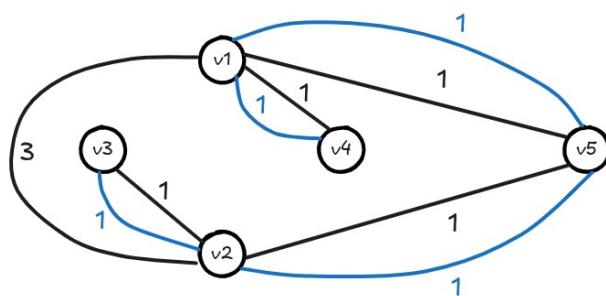
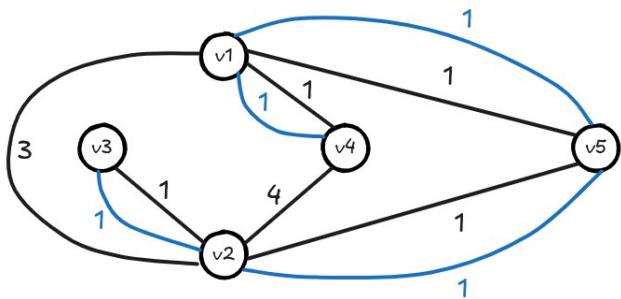
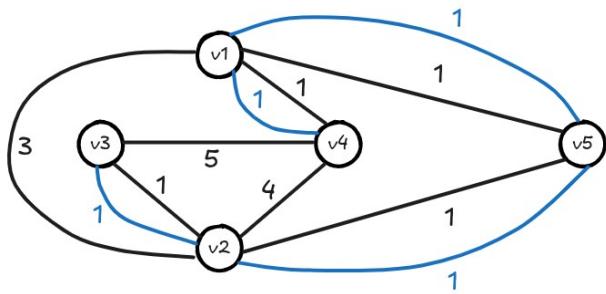
2. Obter o tour de Euler com o algoritmo de Fleury ou Hierholzer

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	$\{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_1, v_4), (v_1, v_5)\}$	v3
2	\emptyset	$\{(v_3, v_2), (v_3, v_2), (v_3, v_4)\}$	v4
3	\emptyset	$\{(v_4, v_1), (v_4, v_1), (v_4, v_2)\}$	v2
4	\emptyset	$\{(v_2, v_1), (v_2, v_3), (v_2, v_3), (v_2, v_5), (v_2, v_5)\}$	v5
5	\emptyset	$\{(v_5, v_2), (v_5, v_1), (v_5, v_1)\}$	v1
6	\emptyset	$\{(v_1, v_2), (v_1, v_4), (v_1, v_4), (v_1, v_5)\}$	v4
7	$\{(v_4, v_1)\}$	\emptyset	v1
8	\emptyset	$\{(v_1, v_2), (v_1, v_5)\}$	v2
9	$\{(v_2, v_5)\}$	$\{(v_2, v_3), (v_2, v_3)\}$	v3
10	$\{(v_3, v_2)\}$	\emptyset	v2
11	$\{(v_2, v_5)\}$	\emptyset	v5
12	$\{(v_5, v_1)\}$	\emptyset	v1

$$T = V_1 V_3 V_4 V_2 V_5 V_1 V_4 V_1 V_5 V_2 V_3 V_2 V_1$$



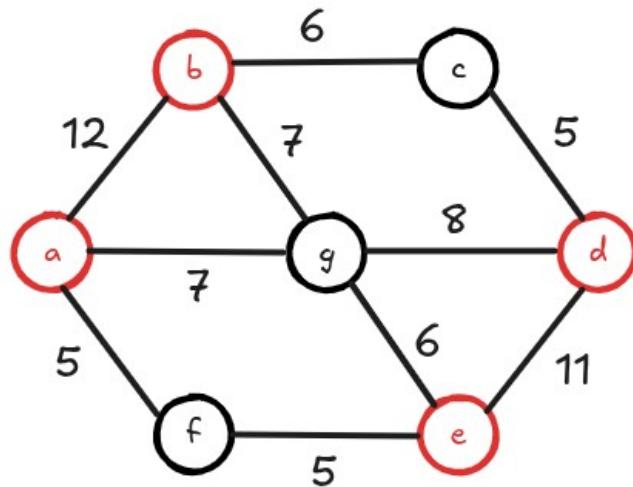
Obs: Cabe ressaltar que num grafo com muitos vértices, há um número imenso de possíveis tours de Euler, mas também há vários pontos em que somos obrigados a evitar pontes para não gerar uma solução inválida.

O caso geral ($|I| > 2$)

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. De maneira resumida a solução envolve os seguintes subproblemas:

- 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: suponha um grafo com $|I| = 4$, ou seja, 4 vértices ímpares.



Note que existem $m = 4$ vértices ímpares: a, b, d, e

Passo 1: Encontrar o caminho mínimo entre todos os vértices ímpares. Isso significa aplicar o algoritmo de Dijkstra diversas vezes. Neste caso, teríamos:

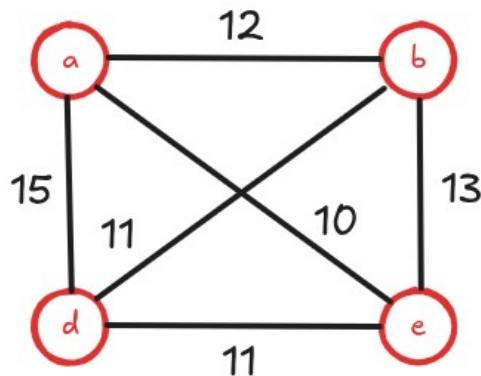
Dijkstra com raiz em a: P_{ab} , P_{ad} , P_{ae}

Dijkstra com raiz em b: P_{bd} , P_{be}

Dijkstra com raiz em d: P_{de} 1

Note que 3 execuções do algoritmo de Dijkstra são suficientes. No caso geral de se ter m vértices ímpares, é necessário executar ao menos $m - 1$ vezes o algoritmo de Dijkstra.

Passo 2: Montar grafo K_m com os vértices ímpares em que os pesos das arestas são os custos dos caminhos mínimos



Passo 3: Obter um emparelhamento perfeito de custo mínimo em K_m (conj independente de arestas)

Note que nesse caso existem apenas 3 possibilidades:

$$(a, b) + (d, e) = 12 + 11 = 23$$

$$(a, d) + (b, e) = 15 + 13 = 28$$

$$(a, e) + (b, d) = 10 + 11 = 21 \leftarrow \text{menor custo}$$

No caso geral, pode-se mostrar que o número de emparelhamentos perfeitos em um grafo completo de m vértices é dado por:

$$\prod_{\substack{i \text{ ímpar} \\ i < m}} i$$

Por exemplo, para os valores de m a seguir temos:

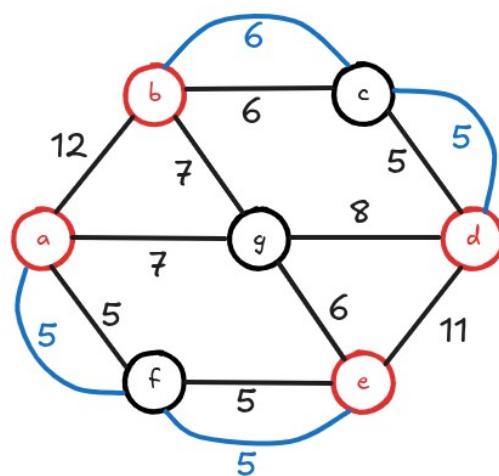
$$m = 4 \rightarrow 1 \times 3 = 3$$

$$m = 6 \rightarrow 1 \times 3 \times 5 = 15$$

$$m = 8 \rightarrow 1 \times 3 \times 5 \times 7 = 105$$

$$m = 10 \rightarrow 1 \times 3 \times 5 \times 7 \times 9 = 945$$

Passo 4: Duplicar em G as arestas dos caminhos mínimos selecionados. Isso garante que não haverá vértices de grau ímpar pois os caminhos mínimos não se cruzam.



Passo 5: Extrair um tour de Euler no grafo resultante aplicando o algoritmo de Fleury ou Hierholzer

$T = a \ b \ c \ d \ g \ b \ c \ d \ e \ f \ a \ g \ e \ f \ a$

Grafos Hamiltonianos

Relação direta com o problema do caixeiro viajante (Traveling Salesman Problem – TSP), que é NP-Completo, sendo um dos mais importantes de toda computação.

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

$$\begin{array}{ccccccc} G & \rightarrow & G' & \rightarrow & G'' & \rightarrow & G''' & \rightarrow \dots & \rightarrow K_n \\ & & +e & & +e & & +e & & +e \end{array}$$

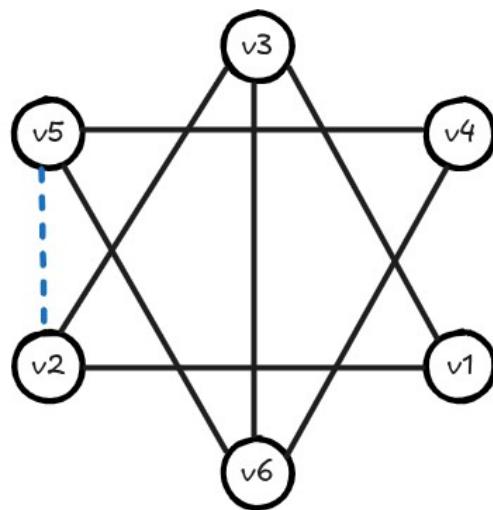
A cada passo, uma aresta é inserida em G. Como K_n é Hamiltoniano, segue que em algum momento entre G e K_n , o grafo torna-se Hamiltoniano. Usaremos essa noção para a definição a seguir.

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)

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: v6 – v4 – v5 – v2 – v1 – v3 – v6

No caso da aresta (v1, v4) temos: v6 – v5 – v4 – v1 – v2 – v3 – v6



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,

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

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)

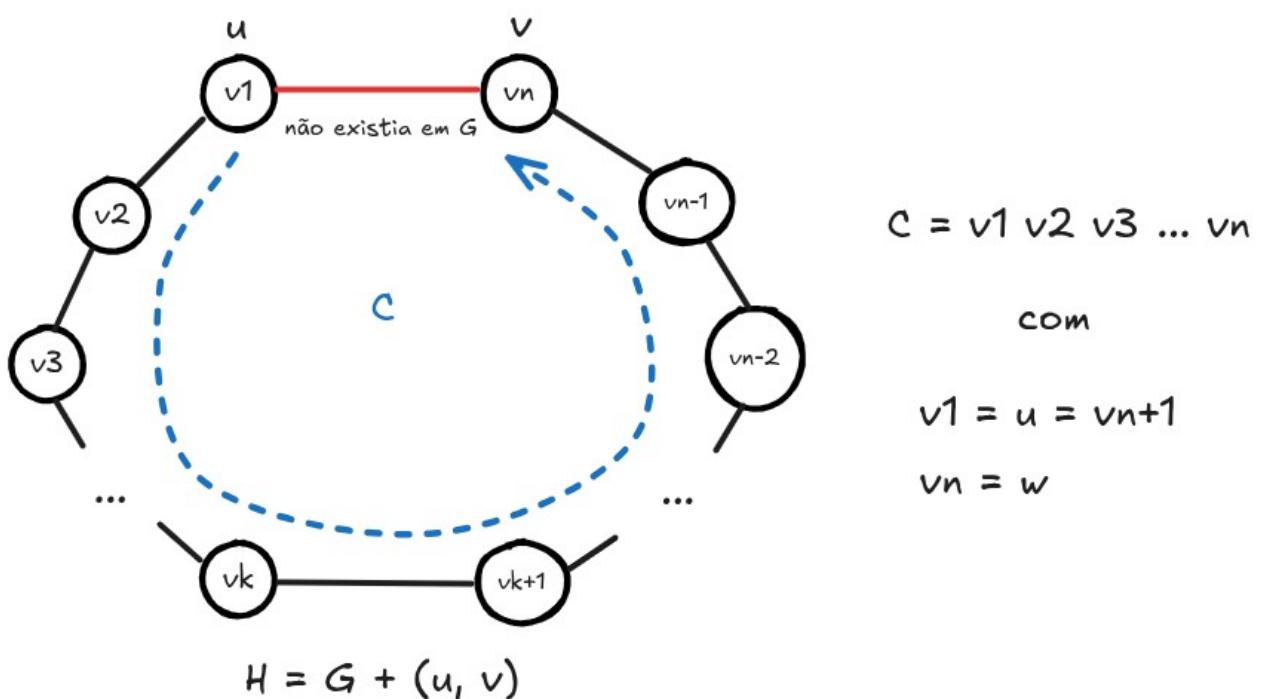
Ideia é verificar que não pode existir G que satisfaça propriedade H e não seja Hamiltoniano.

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 $u, w \in V$ não adjacentes.

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

Chamaremos $u = v_1$ e $w = v_n$.



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!

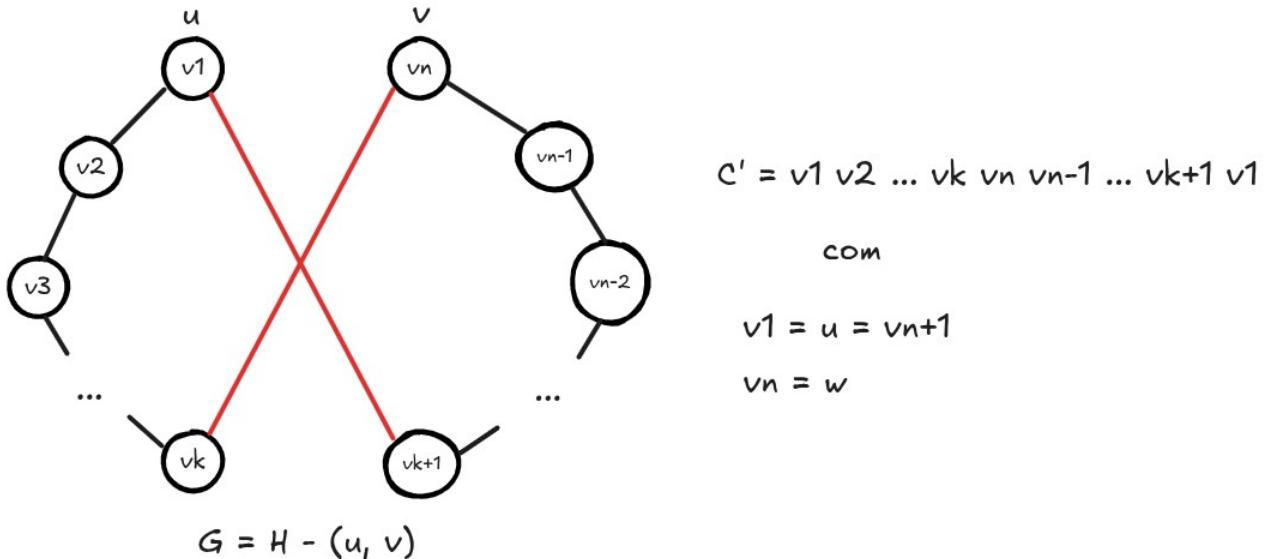
ii) $v_n \notin T$: Porque? \exists aresta que liga w a $v_n = w$? Não pois não há loops!

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



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, da teoria dos conjuntos 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:

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

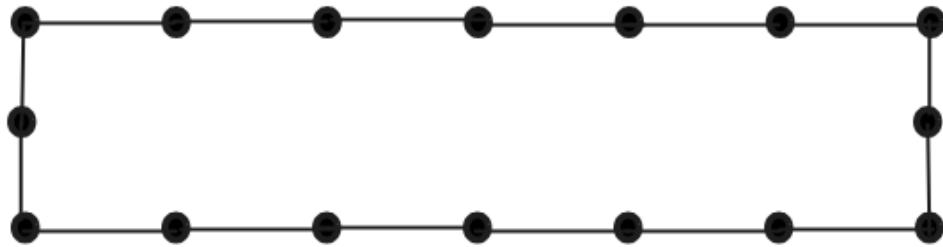
o que é uma contradição para a primeira suposição, pois todo grafo que satisfaz H tem

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

para quaisquer u e w em V, ou seja, $d(u) + d(w) \geq n$. Em outras palavras, a hipótese de que existe G que satisfaz H e não seja Hamiltoniano é falsa.

Conclusão: Não existe grafo G que satisfaz a propriedade H e 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 podemos 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 teorema anterior, apesar de poderoso, não nos fornece uma caracterização completa de grafos Hamiltonianos. Basicamente, ele diz que todo grafos que satisfaz a propriedade H é Hamiltoniano, mas nem todo grafo Hamiltoniano satisfaz a propriedade H .

Pergunta: Você é capaz de apontar um grafo Hamiltoniano que não satisfaz H ?



Teorema (Ore): Seja $G = (V, E)$ um grafo básico simples de n vértices e sejam u e v vértices não adjacentes tais que:

$$d(u) + d(v) \geq n$$

Então, G é Hamiltoniano se e somente se $G + (u, v)$ é Hamiltoniano.

A. (ida) $p \rightarrow q$

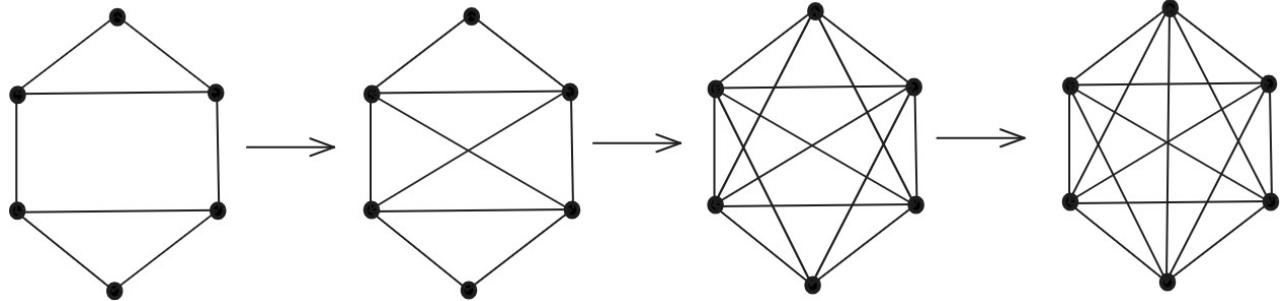
Se G é Hamiltoniano, é trivial que a adição de qualquer aresta faz com que ele permaneça Hamiltoniano.

B. (volta) $q \rightarrow p$ (por contradição)

1. Suponha que $H = G + (u, v)$ seja Hamiltoniano.
2. Então, para G não ser Hamiltoniano devemos ter $d(u) + d(v) < n$ (contrapositiva do teorema de Dirac)
3. Essa é uma contradição para a hipótese inicial.
4. Portanto, G deve ser Hamiltoniano.

Def: Fechamento de um grafo G .

Seja $G = (V, E)$ um grafo básico simples. Enquanto existir vértices não adjacentes u e v tal que $d(u) + d(v) \geq n$, una-os por uma aresta. O grafo final, $c(G)$, é denominado de fechamento de G .



Teorema (Bondy e Chvatal, 1976): Um grafo básico simples G é Hamiltoniano se e somente se seu fechamento $c(G)$ é Hamiltoniano.

A. (ida) $p \rightarrow q$

É fácil perceber que se G é Hamiltoniano, a adição de qualquer aresta em G o mantém Hamiltoniano.

B. (volta) $q \rightarrow p$

1. Suponha que $c(G)$ seja Hamiltoniano.

2. Seja $G, G_1, G_2, G_3, \dots, G_{K-1}, G_K = c(G)$ a sequência de grafos obtidos aplicando a operação de fechamento.

3. Como $c(G) = G_K$ é obtido de G_{K-1} por

$$G_K = G_{K-1} + (u, v)$$

onde u, v são vértices não adjacentes em G_{K-1} com $d(u) + d(v) \geq n$.

4. Pelo teorema anterior, G_{K-1} é Hamiltoniano.

5. Pelo mesmo raciocínio, $G_{K-2}, G_{K-3}, \dots, G$ são Hamiltonianos.

Corolário: Seja $G = (V, E)$ um grafo básico simples com $|V| > 2$. Se $c(G) = K_n$, ou seja, o fechamento é Hamiltoniano, então G é Hamiltoniano.

Teorema (Pósa, 1962): Seja $G = (V, E)$ um grafo básico simples com $|V| > 2$ e $d_1, d_2, d_3, \dots, d_n$ a sequência dos graus dos vértices em ordem crescente. Se para todo i , tal que $1 \leq i \leq \frac{n}{2}$, $d_i \geq i+1$, então G é Hamiltoniano.

Por questões de complexidade, iremos omitir a prova desse importante resultado.

Ex: $L_G = (2, 3, 4, 4, 5, 5, 5, 6)$ é a lista de graus de um grafo G Hamiltoniano? Justifique.

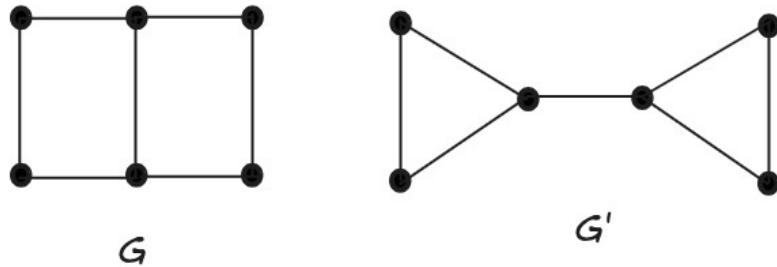
$$n/2 = 4 \quad i = 1, 2, 3, 4$$

i = 1:	$d_1 = 2 \geq 1+1$	ok
i = 2:	$d_2 = 3 \geq 2+1$	ok
i = 3:	$d_3 = 4 \geq 3+1$	ok
i = 4:	$d_4 = 4 \geq 4+1$	X (falhou)

Portanto, não podemos afirmar que G é Hamiltoniano (pode ser ou pode não ser: indeterminação).

Observação: Note que uma mesma sequência de graus pode representar dois grafos distintos: um grafo G Hamiltoniano e um grafo G' não Hamiltoniano.

$$L = (2, 2, 2, 2, 3, 3)$$



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

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. É um problema de otimização e não de decisão (NP-Hard).

Pergunta: como obter um ciclo Hamiltoniano de custo mínimo?

- Não se conhece algoritmo ótimo (não há garantias de que termos a solução ótima x^*)
- Devemos trabalhar com algoritmos aproximados (sub-ótimos).

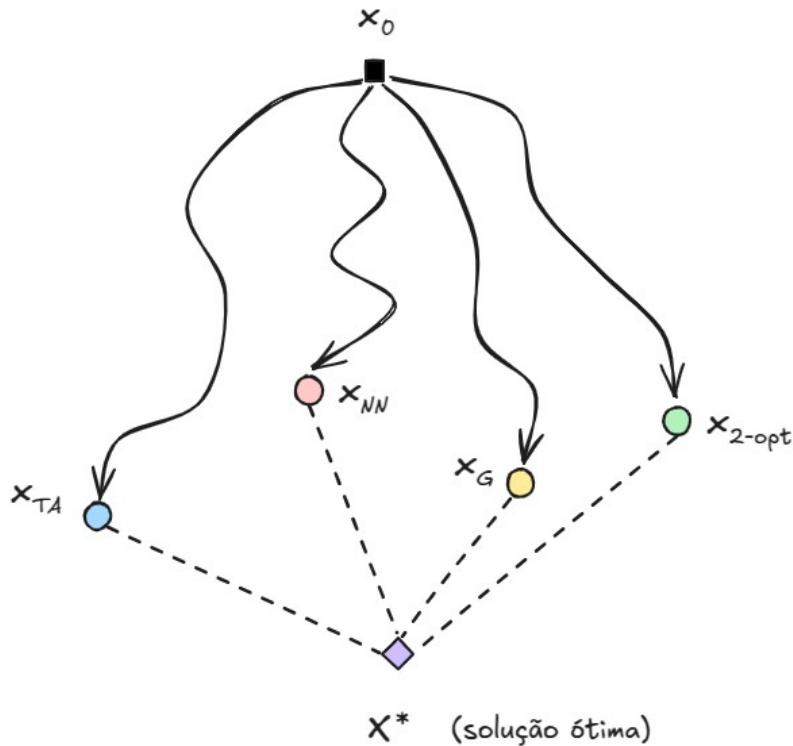
Mas como medir a qualidade da solução obtida por um algoritmo aproximado?

Devemos medir quão próximo da solução ótima é a solução aproximada. 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 $f(x_A) \leq c f(x^*)$



TSP métrico

Seja $G = (V, E, w)$ um grafo conexo com $w: E \rightarrow R^+$, em que w é uma função de distância. Então, a desigualdade triangular é satisfeita, ou seja:

$$\forall i, j, k \quad w(v_i, v_k) \leq w(v_i, v_j) + w(v_j, v_k)$$

Isso significa que tomar atalhos é sempre vantajoso (corta caminhos).

Uma instância do TSP em que os pesos das arestas do grafo de entrada G é uma função de distância é conhecida como TSP métrico.

Algoritmos para o TSP

A seguir apresentamos alguns dos principais algoritmos aproximados para o TSP, iniciando pelo método dos vizinhos mais próximos.

Nearest Neighbor

O algoritmo mais simples para o TSP utiliza uma abordagem gulosa: visitar sempre o vértice mais próximo do vértice atual. Porém, esse algoritmo não fornece boas soluções, em especial quando o número de vértices cresce de maneira arbitrária.

```
Nearest_Neighbor(G, w, v0) {
    H = [v0]
    while |H| < n {
        x = Last(H)      # retorna o último vértice de H
```

```

    Defina  $v_i$  como o vizinho mais próximo de  $x$ 
    InsertEnd(H,  $v_i$ )      # insere  $v_i$  no final de H
}
InsertEnd(H,  $v_0$ )
}

```

Teorema: A solução obtida pelo algoritmo Nearest Neighbor no TSP métrico satisfaz:

$$f(x_{NN}) \leq \frac{1}{2}(\log_2 n + 1)f(x^*)$$

Note que se o número de vértices do grafo G aumenta, a qualidade da solução cai de maneira logarítmica. Em outras palavras, não é muito bom para grafos com grande quantidade de vértices.

Análise da complexidade

1. Note que o loop WHILE executa n vezes

2. Obter o último elemento da lista H pode ser realizado em tempo constante $O(1)$ se mantivermos um contador para o número de elementos em H .

3. Para definir o vizinho mais próximo de x , é preciso verificar todas as distâncias da linha de x na matriz de adjacências, o que é $O(n)$

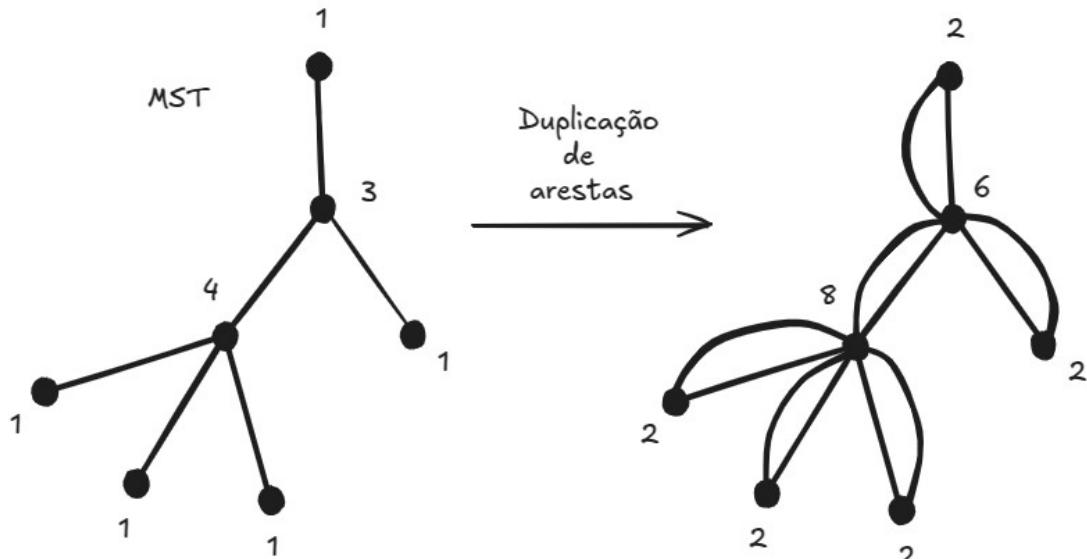
4. Inserção no final da lista H também pode ser feito em $O(1)$ se mantivermos um contador para o número de elementos de H

Custo total: $n \times [O(1) + O(n) + O(1)] = O(n^2)$

Twice-Around

Este algoritmo combina a árvore geradora mínima de G e circuitos Eulerianos para construir uma solução aproximada para o TSP métrico.

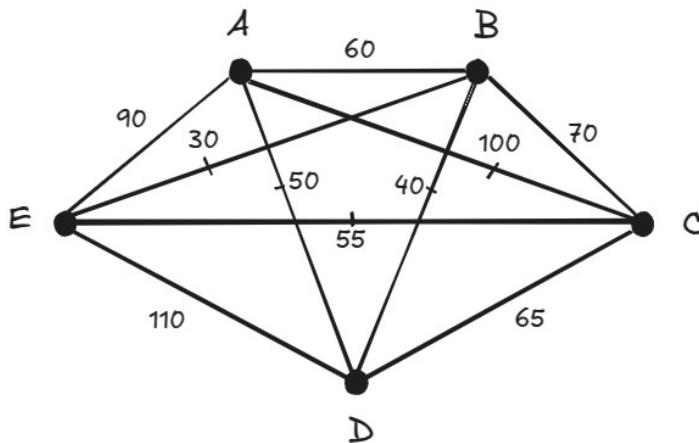
Teorema: Seja $G = (V, E)$ um grafo conexo. Se T é uma MST de G , então se definirmos um supergrafo T' duplicando toda aresta de T , T' será Euleriano.



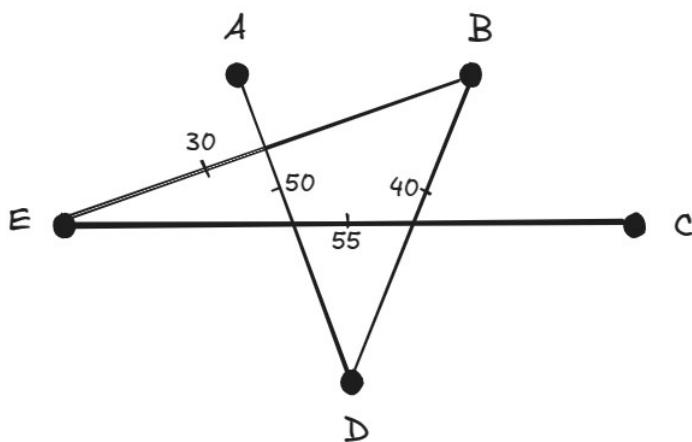
É fácil notar que ao duplicar todas as arestas da MST, estamos multiplicando os graus por 2. Assim, os nós folhas, passarão a ter grau 2 e todos os demais nós internos serão múltiplos de 2. A seguir apresentamos o algoritmo Twice_Around, para a solução aproximada do TSP métrico.

```
Twice_Around(G, w, v0) {
    H = ∅
    T = MST(G)           # Extrai MST de G
    for each e ∈ T       # Duplica as arestas da MST
        T = T ∪ {e}
    L = Eulerian_Circuit(T)   # Obtém circuito Euleriano
    while L ≠ ∅ {
        l = RemoveFirst(L)
        if l ∉ H          # Remove as repetições
            H = H ∪ {l}
    }
    return H
}
```

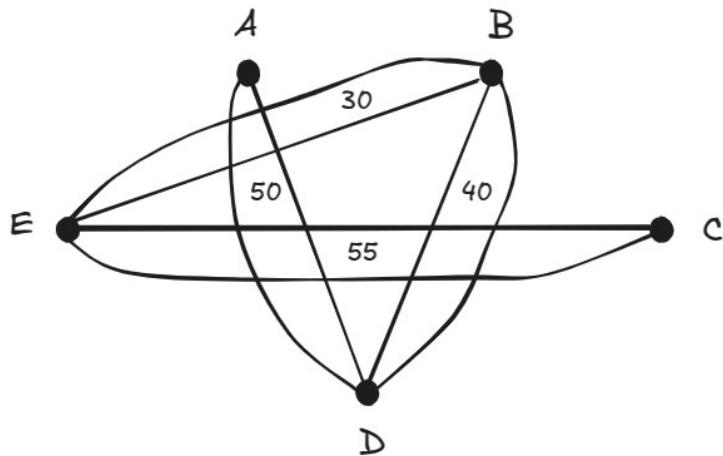
Veremos a seguir um exemplo ilustrativo da aplicação do algoritmo Twice-Around.



1º passo: Extrair MST de G



2º passo: Duplicar arestas da MST

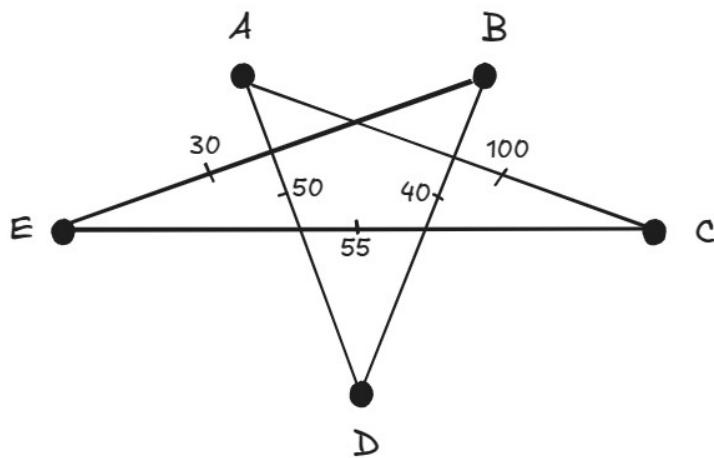


3º passo: Extrair um circuito Euleriano L

$$L = [A, D, B, E, C, E, B, D, A]$$

4º passo: Eliminar as repetições (equivale a tomar atalhos)

$$H = [A, D, B, E, C, A]$$



Note que o custo do ciclo Hamiltoniano obtido é $w(C) = 50 + 40 + 30 + 55 + 100 = 275$

Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (estruturas estáticas) ou $O(m \log n)$ (estruturas dinâmicas).
2. A duplicação das arestas da MST pode ser feita em $O(m)$
3. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$
4. Por fim, o WHILE é executado uma vez para cada aresta duplicada: 2m iterações, o que é $O(m)$

5. Portanto, o custo final é: $O(m \log n) + O(m) + O(m) + O(m) = O(m \log n)$

Ou seja, o ponto crítico (gragalo) no algoritmo Twice-Around é o cálculo da MST.

Veremos a seguir um resultado muito importante sobre a qualidade da solução obtida pelo algoritmo Twice-Around.

Teorema: A solução obtida pelo algoritmo Twice-Around no TSP métrico satisfaz:

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

ou seja, a solução do TA é no máximo 2 vezes pior que a solução ótima (não importa o número de vértices de G). Melhor que Nearest Neighbor.

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 (**)

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

O algoritmo de Christofides

A ideia do algoritmo de Christofides consistem em melhorar ainda mais o algoritmo Twice-Around, modificando a forma com que transformamos a MST de G em um grafo Euleriano. Ao invés de simplesmente duplicar as arestas da árvore, devemos encontrar um emparelhamento perfeito de custo mínimo entre os vértices de grau ímpar na MST. Dessa forma, com menos duplicações de arestas, o custo adicional é menor.

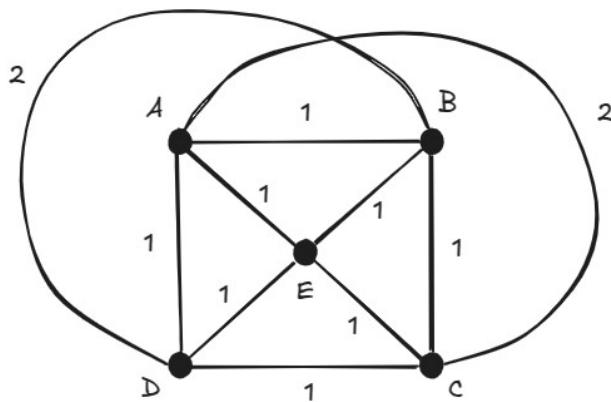
```
Christofides(G, w, v0) {
    H = ∅
    T = MST(G) # Extrai MST de G
    K = Odd_Vertices(T) # Constrói Kn (ímpares)
    M = Minimum_Weight_Matching(Ku) # Emparelhamento mínimo
```

```

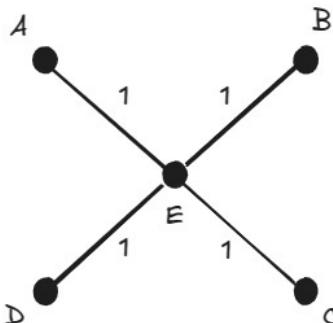
T = T + M           # Adiciona arestas de M
T = Eulerian_Circuit(T)   # Obtém circuito Euleriano
while L ≠ ø {
    l = RemoveFirst(L)
    if l ∉ H           // Remove as repetições
        H = H ∪ {l}
}
return H
}

```

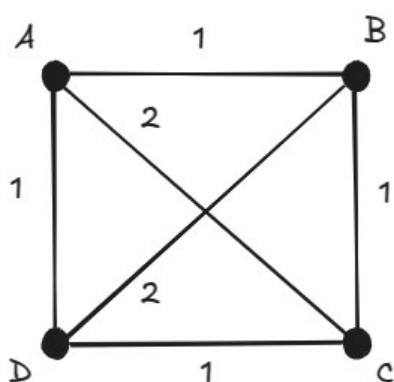
A seguir veremos um exemplo ilustrativo do funcionamento do algoritmo de Christofides. Considere o seguinte grafo G.



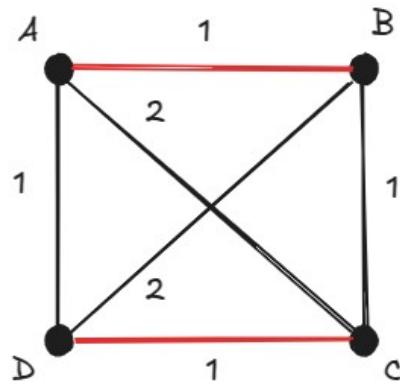
1º passo: Extrair a MST de G



2º passo: criar grafo completo K_n com todos os vértices de grau ímpar e ponderar as arestas com os caminhos mínimos dentre cada par de vértices.



3º passo: encontrar um emparelhamento de custo mínimo entre os vértices de grau ímpar.



Pode-se mostrar que o número de emparelhamentos perfeitos no K_n é dado por:

$$\prod_{i \text{ ímpar}, i < n} i$$

No caso de $n = 4$, temos $1 \times 3 = 3$

No caso de $n = 6$, temos $1 \times 3 \times 5 = 15$

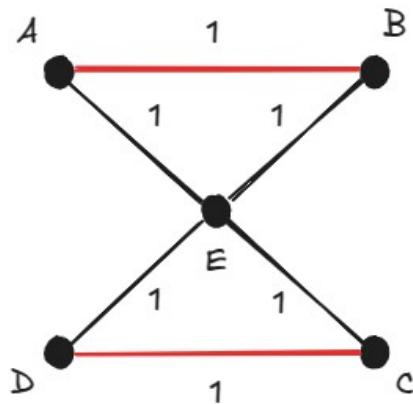
No caso de $n = 8$, temos $1 \times 3 \times 5 \times 7 = 105$

No caso de $n = 10$, temos $1 \times 3 \times 5 \times 7 \times 9 = 945$

Note que o número de possíveis emparelhamentos perfeitos cresce exponencialmente, o que inviabiliza a busca exaustiva pelo emparelhamento mínimo quando o número de vértices cresce. Existe um algoritmo com complexidade $O(n^2 m)$ que obtém o emparelhamento M de custo mínimo em um grafo arbitrário (não precisa ser bipartido). É o algoritmo de Edmonds, também conhecido como *Blossom algorithm*. Para os leitores interessados, indicamos o link a seguir:

https://en.wikipedia.org/wiki/Blossom_algorithm

4º passo: Adicione as arestas de M à MST



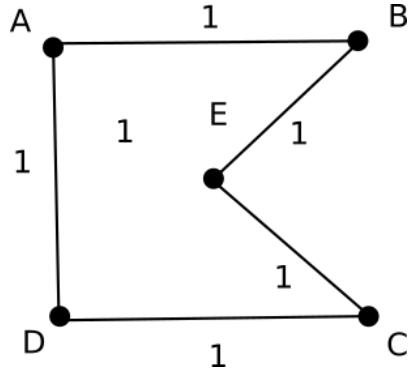
Agora, todos os vértices possuem grau par, então é possível extrair um circuito Euleriano.

5º passo: Encontre um circuito Euleriano L

$L = [A, B, E, C, D, E, A]$

6º passo: Eliminar as repetições (equivalente a tomar atalhos)

$L = [A, B, E, C, D, A]$



Análise da complexidade

1. Obter a MST de G com Kruskal é $O(m \log n)$. Com Prim é $O(n^2)$ (matriz de adjacências + array) ou $O(m \log n)$ (listas de adjacências + min-heap).

2. Obter o emparelhamento de custo mínimo M entre os vértices de grau ímpar: seja z o número de vértices ímpares. O grafo completo formado pelos vértices ímpares, denotado por K_z , terá m' arestas, onde:

$$m' = \frac{z(z-1)}{2}$$

3. Para ponderar os pesos do grafo K_z , será preciso encontrar caminhos mínimos entre cada par de vértices. Podemos executar z vezes o algoritmo de Dijkstra no grafo original, o que nos leva a uma complexidade $O(z m \log n)$

4. Para a construção do emparelhamento M de mínimo custo, o algoritmo de Edmonds possui complexidade $O(z^2 m')$

5. A extração do circuito Euleriano L é lo algoritmo de Hierholzer é $O(m)$

6. Por fim, o WHILE é executado uma vez para cada aresta duplicada: $2m$ iterações, o que é $O(m)$

7. Portanto, o custo final é:

$$O(m \log n) + O(z m \log n) + O(k^2 m') + O(m) \rightarrow O(z m \log n) + O(z^2 m')$$

Ou seja, é quadrático no número de vértices ímpares! Quando o número de vértices ímpares é grande, ou seja, $z \rightarrow n$, o custo pode ser bastante elevado em comparação com o algoritmo Twice-Around.

A seguir, iremos demonstrar um resultado fundamental sobre o algoritmo de Christofides.

Teorema: O algoritmo de Christofides fornece uma $\frac{3}{2}$ -aproximação para o TSP métrico.

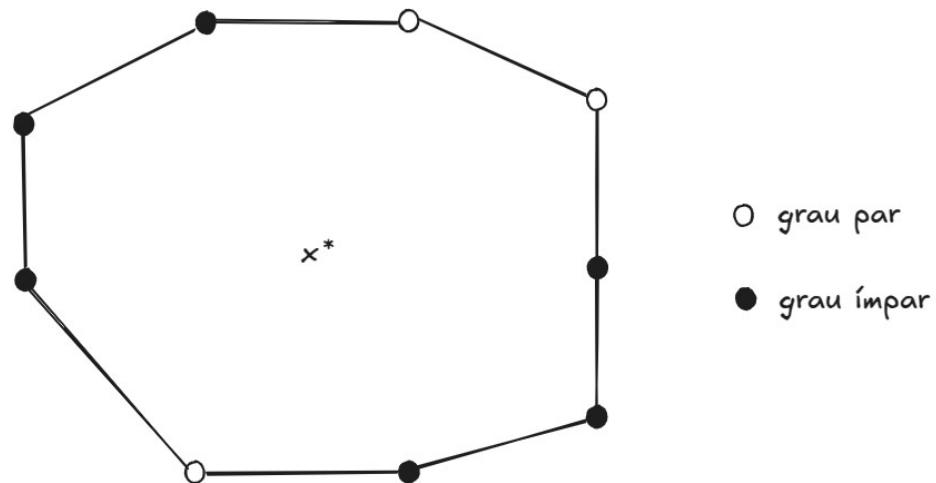
Prova: Desejamos mostrar que:

$$f(x_c) \leq \frac{3}{2} f(x^*)$$

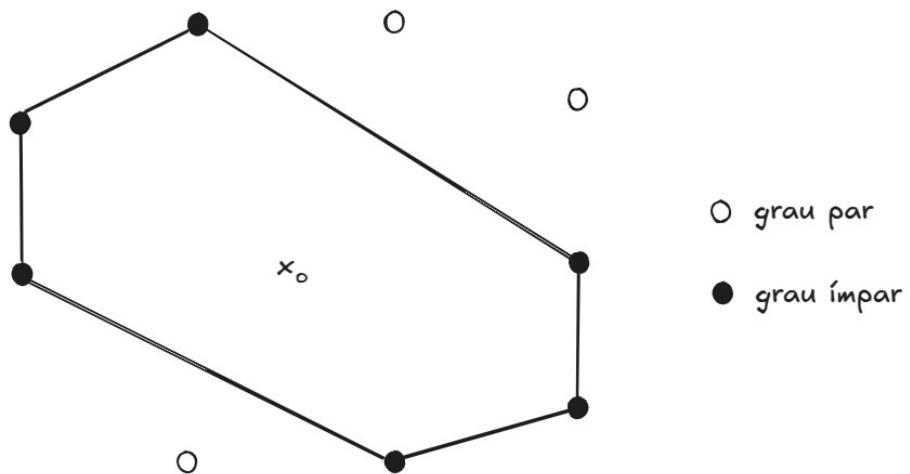
onde x^* é a solução ótima. Sabemos que o custo da MST T é menor que x^* , ou seja:

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

pois $x^* - e$ é uma árvore com peso maior ou igual ao peso da MST de G . Basta agora provar que o custo do emparelhamento M entre os vértices ímpares é menor ou igual a $\frac{1}{2} f(x^*)$. Lembre-se que aqui, ao invés de duplicar todas as arestas da MST, adicionamos menos arestas a MST. Iniciamos a discussão com o ciclo Hamiltoniano de custo mínimo, cujo custo é $f(x^*)$.



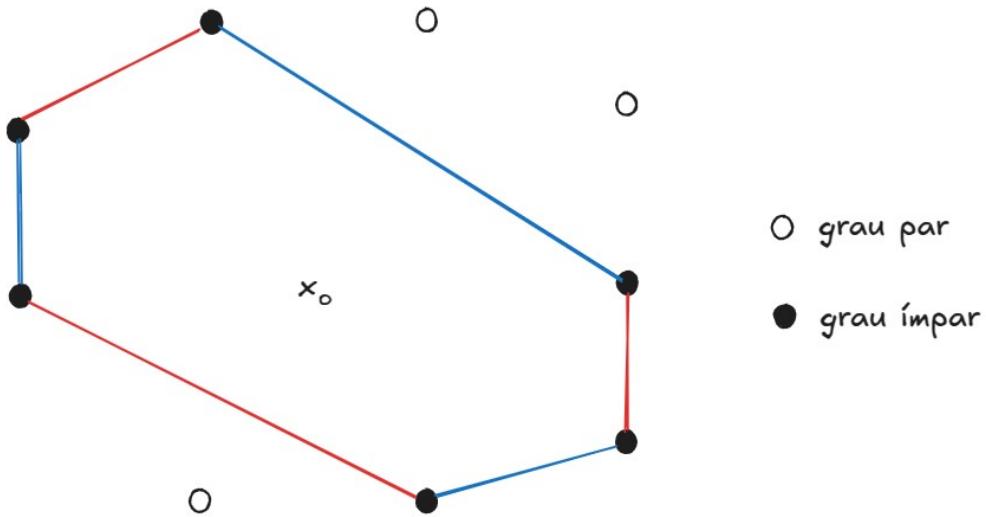
onde os vértices pretos denotam os vértices de grau ímpar (subconjunto O de odd degree) e os vértices brancos denotam os vértices de grau par. Existe um ciclo Hamiltoniano de custo mínimo que passa apenas pelos vértices do subconjunto O , denotado por x_o , conforme ilustra a figura a seguir:



O custo do ciclo x_o é menor que o custo da solução ótima, pois ele passa por menos vértices:

$$f(x_o) \leq f(x^*)$$

Porém, note que podemos particionar o ciclo x_o em arestas alternadas de duas cores: vermelhas e azuis. Note que como o número de vértices ímpares é sempre par, o número de arestas no ciclo x_o será sempre par, e assim, temos 2 emparelhamentos distintos com o mesmo número de arestas: M' (azuis) e M'' (vermelhas).



Como o peso total dos 2 emparelhamentos é igual ao custo do ciclo x_o , temos:

$$w(M') + w(M'') = f(x_o) \leq f(x^*)$$

Se os pesos de M' e M'' forem iguais:

$$w(M') \leq \frac{f(x^*)}{2}$$

Se os pesos de M' e M'' forem diferentes, seja $\bar{M} = \min\{M', M''\}$. Então, temos:

$$w(\bar{M}) \leq \frac{f(x^*)}{2}$$

Assim, como no algoritmo de Christofides encontramos o emparelhamento M de custo mínimo entre os vértices de grau ímpar, $M = \bar{M}$ e $w(M) \leq \frac{f(x^*)}{2}$. Portanto o custo da solução final satisfaz:

$$f(x_c) = w(T_{MST}) + w(M)$$

Como $w(T_{MST}) < f(x^*)$ e $w(M) \leq \frac{f(x^*)}{2}$, temos finalmente:

$$f(x_c) \leq f(x^*) + \frac{f(x^*)}{2} = \frac{3}{2}f(x^*)$$

O algoritmo 2-optimal

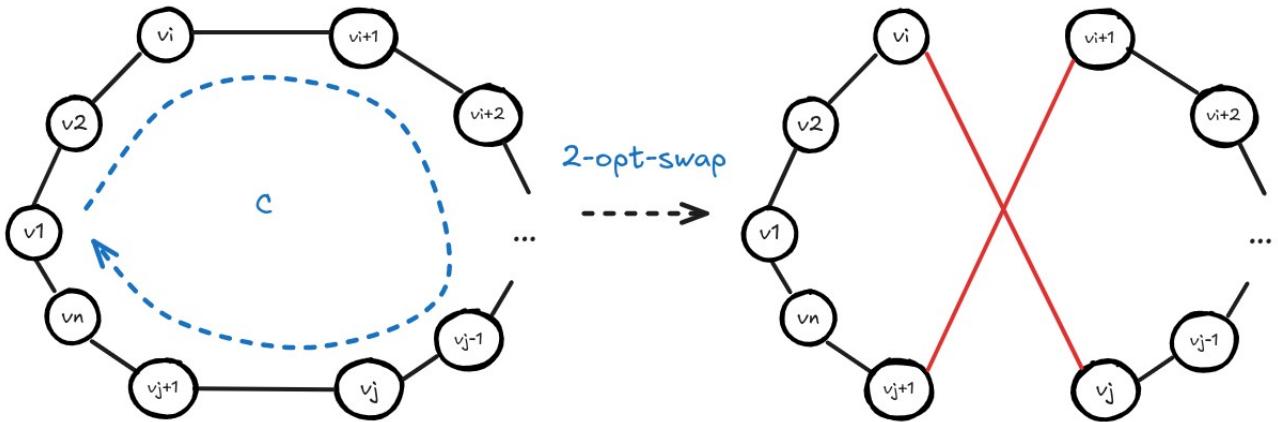
Ideia: partir de uma solução inicial arbitrária e tentar melhorá-la iterativamente a partir de buscas na vizinhança local, através de operações de religações.

Pseudocódigo

Sem perda de generalidade, seja $C = v_1 v_2 v_3 \dots v_{n-1} v_n$ a solução inicial. Definiremos a operação Two_Opt_Swap(C) conforme a seguir:

```
Two-Opt-Swap( $C$ ,  $i$ ,  $j$ ) {  
    Seja  $C_{ij}$  p ciclo obtido pela religação de  $v_i$  e  $v_j$  como segue:
```

$$C' = v_1 v_2 \dots v_i v_j v_{j-1} v_{j-2} \dots v_{i+1} v_{j+1} \dots v_n v_1$$



```
    return  $C'$   
}
```

Essa operação é a base para a busca local e define como perturbamos a solução corrente na busca por uma solução melhor que esteja nas proximidades.

Note que em essencia, o que a operação Two-Opt-Swap faz é primeiramente desconectar v_i de v_{i+1} e desconectar v_j de v_{j+1} e em seguida conectar com uma aresta o par de vértices v_i e v_j , além de conectar com outra aresta o par de vértices v_{i+1} e v_{j+1} . A vantagem desta operação é que ela é eficiente, uma vez que pode ser realizada com complexidade constante $O(1)$.

A seguir é apresentado o algoritmo 2-Optimal, também conhecido como 2-Opt, que é um método iterativo para melhorar uma solução inicial dada como entrada.

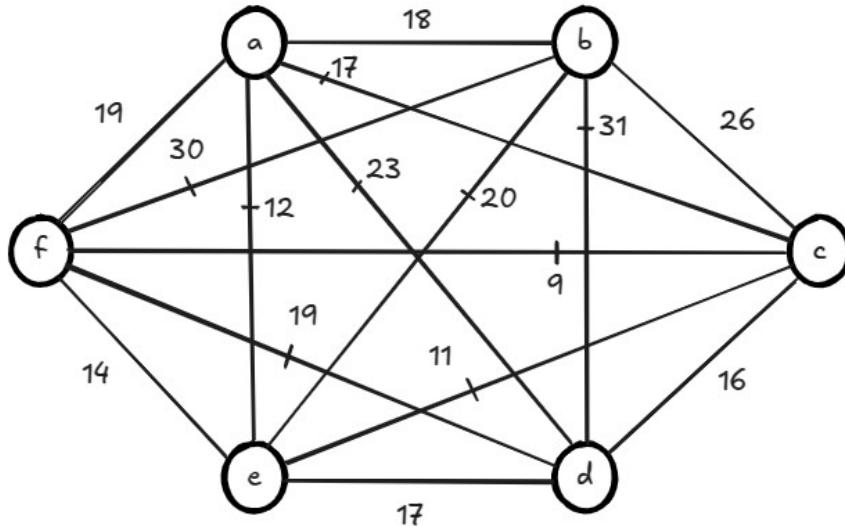
```
Two-Opt( $C$ ,  $n$ ) {  
    best =  $C$   
    improved = True  
    while improved {  
        improved = False  
        for  $i$  = 1 to  $n-2$  {  
            for  $j$  =  $i+2$  to  $n$  {  
                 $C$  = Two-Opt-Swap( $C$ ,  $i$ ,  $j$ )  
                if  $w(C) < w(best)$  {  
                    best =  $C$ 
```

```

        improved = True
    }
}
C = best
}
}

```

A seguir mostramos um exemplo ilustrativo da aplicação do algoritmo 2-Opt

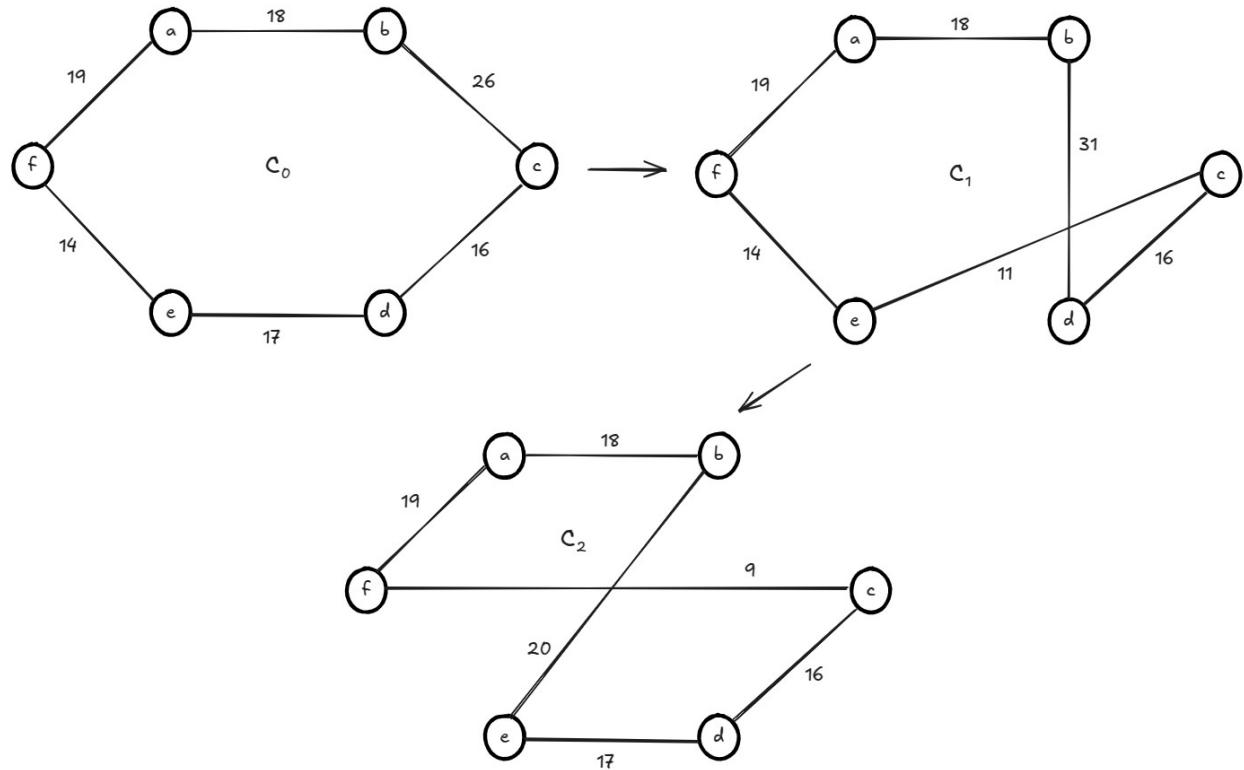


Consideramos como solução inicial o ciclo $\mathbf{C} = \mathbf{a} \mathbf{b} \mathbf{c} \mathbf{d} \mathbf{e} \mathbf{f} \mathbf{a}$ que possui custo $w(\mathbf{C}) = 110$.

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

Após toda uma varredura ao redor da solução com custo $w(C) = 99$, não foi possível encontrar nenhuma solução melhor. PARE!



Análise da complexidade

É possível realizar a operação Two_Opt_Swap(C, i, j) com complexidade O(1). Sendo assim, em uma rodada do algoritmo 2-Opt, temos o seguinte número de operações:

$$T(n) = \sum_{i=1}^{n-2} \sum_{j=i+2}^n 1 = (n-2) + (n-3) + (n-4) + \dots + (n-(n-1)) = (n+n+n+\dots+n) - (2+3+4+\dots+(n-1))$$

Note que o primeiro somatório é justamente $n(n-2)$. Já o segundo somatório é igual a:

$$\left(\sum_{i=1}^{n-1} i \right) - 1 = \frac{n(n-1)}{2} - 1 = \frac{n^2 - n - 2}{2}$$

Logo, temos:

$$T(n) = n^2 - 2n - \left(\frac{n^2}{2} - \frac{n}{2} - 1 \right) = \frac{n^2}{2} - \frac{3n}{2} + 1 = \frac{n^2 - 3n + 2}{2}$$

o que é $O(n^2)$. Portanto, supondo que o loop WHILE mais externo execute K vezes, teremos uma complexidade $O(Kn^2)$. Como K << n (K é uma constante muito menor que n), podemos dizer que a complexidade do algoritmo 2-Opt é $O(n^2)$.

Teorema: A solução obtida pelo algoritmo 2-Opt em uma instância do TSP métrico satisfaz:

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

Iremos omitir a prova deste resultado, mas de qualquer forma, note que a qualidade da solução depende do número de vértices do grafo. Quanto mais vértices o grafo possui, menos garantias de que estamos obtendo soluções próximas à solução ótima.

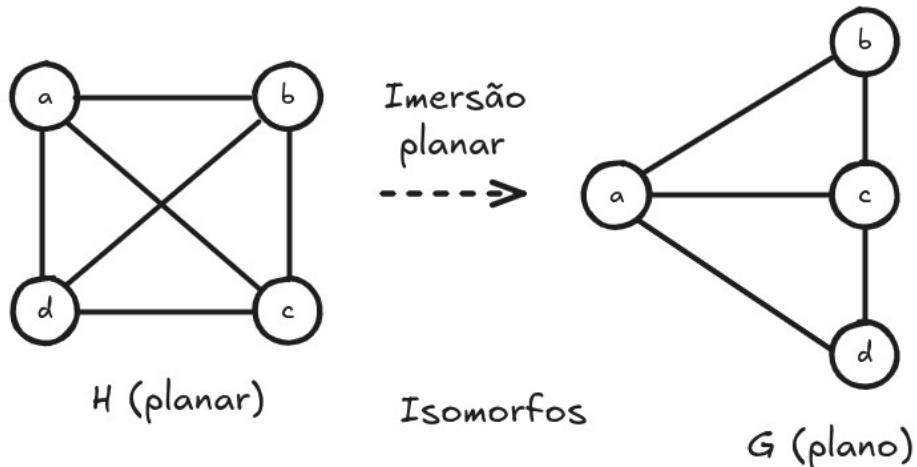
Uma estratégia comum consiste em utilizar os algoritmos Nearest Neighbor, Twice-Around e Christofides para gerar soluções iniciais para o 2-Opt, na tentativa de melhorá-las ao menos um pouco.

Existem algoritmos exatos para o TSP: um deles é o método da força bruta, que testa todas as combinações possíveis de ciclos de comprimento n e possui complexidade $O(n!)$, sendo inviável para a maioria dos problemas. O algoritmo de Held-Karp utiliza uma abordagem baseada em programação dinâmica para resolver o TSP e possui complexidade $O(2^n n^2)$ (exponencial), o que ainda é inviável para grafos com um grande número de vértices. Por essa razão, em geral, utiliza-se os algoritmos aproximados na solução de problemas reais.

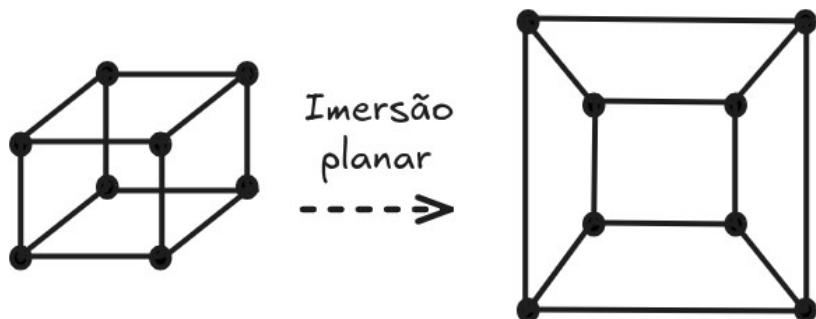
Grafos Planares

Grafos planares são um conceito fundamental na teoria dos grafos, referindo-se a grafos que podem ser desenhados em um plano (ou uma esfera) de forma que nenhuma de suas arestas se cruze, exceto nos vértices. Essa propriedade tem implicações profundas em diversas áreas, desde o design de circuitos integrados e mapas geográficos até a otimização de redes de comunicação. A importância dos grafos planares reside na sua capacidade de modelar problemas do mundo real onde a não intersecção física é crucial, facilitando o entendimento de como sistemas podem ser organizados de maneira eficiente e sem sobreposições indesejadas. Seu estudo leva a teoremas clássicos como a fórmula de Euler para poliedros e o critério de Kuratowski, que fornecem ferramentas poderosas para determinar a planaridade de um grafo e analisar suas propriedades topológicas.

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 não importa, apenas topologia.



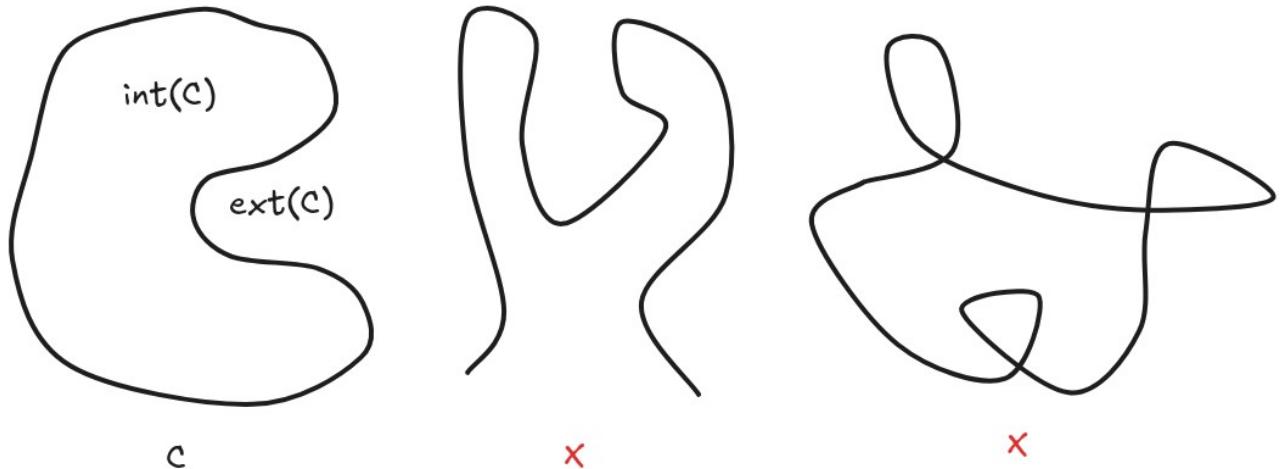
Importância:

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

Precisamos de uma maneira objetiva de caracterizar planaridade.

Def (Curva de Jordan):

Toda plana curva fechada que não intercepta a si própria.



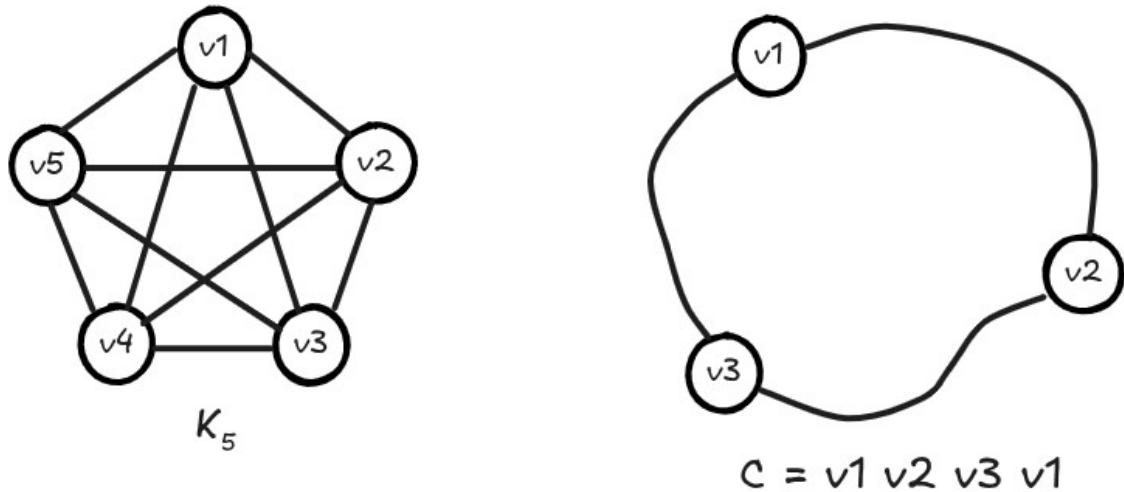
Obs: A noção de curva de Jordan para nós em grafos será a de ciclo. Todo ciclo pode ser visto como uma curva de Jordan uma vez que é uma trajetória fechada e que não repete vértices

Teorema: 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 $C = v1 v2 v3 v1$ (curva de Jordan)

Como $v_4 \notin C$ temos 2 opções:

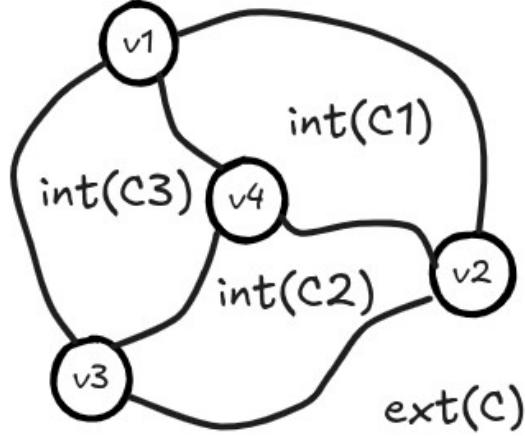
- a) $v_4 \in \text{int}(C)$
- b) $v_4 \in \text{ext}(C)$

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

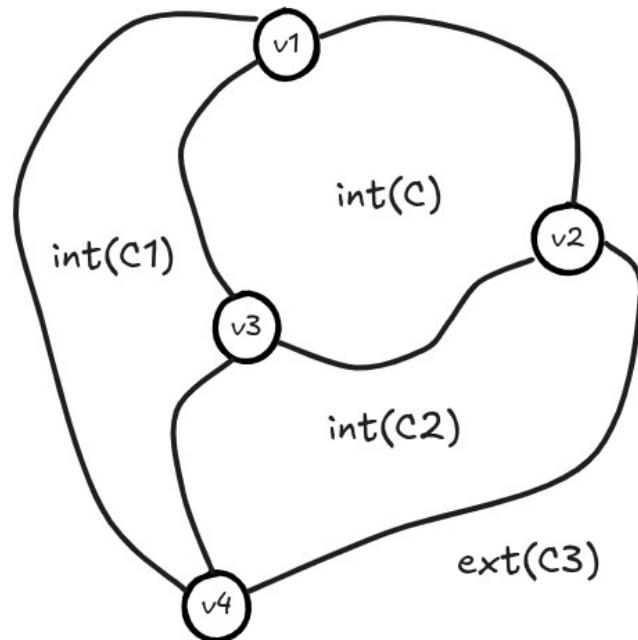
Senso assim, para v_4 no interior de C , não é possível K_5 ser planar. Vamos agora ao caso b), onde $v_4 \in \text{ext}(C)$. Assim, temos:

$$C = v_1 v_2 v_3 v_1$$

$$C_1 = v_1 v_3 v_4 v_1$$

$$C_2 = v_2 v_3 v_4 v_2$$

$$C_3 = v_1 v_2 v_4 v_1$$



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.

Teorema (Fórmula de Euler): Seja $G = (V, E)$ um grafo plano conexo com $|V| = n$, $|E| = m$ e f faces ou regiões. Então,

$$n - m + f = 2$$

Prova (por indução em m):

Base: ($m = 0$)

$P(0)$: G é o grafo nulo (sem arestas). Nesse caso, $n = 1$ e $f = 1$, o que nos leva a

$$1 - 0 + 1 = 2 \quad (\text{ok})$$

Passo de indução: $P(k) \rightarrow P(k+1)$ para k arbitrário

Hipótese de indução $P(k)$: $n - k + f = 2$ (assume-se que isso vale para todo grafo G com k arestas e iremos provar que se isso vale então ao adicionar uma nova aresta vai continuar valendo).

1. Suponha G em que $m = k + 1$ (o número de arestas é exatamente uma unidade maior).

2. Se G é uma árvore, então $m = n - 1$, o que implica em $n = m + 1 = k + 2$.

a) Como não há ciclos em árvores, existe uma única face, ou seja, $f = 1$.

b) Assim, temos $n - m + f = (k + 2) - (k + 1) + 1 = 2 - 1 + 1 = 2$ (ok, temos que $P(k+1)$ é V)

3. Se G não é uma árvore, então G contém um ciclo C .

a) Seja e uma aresta do ciclo C . Faça $G' = G - e$.

b) G' tem $m' = k = m - 1$ arestas e $n' = n$ vértices.

4. Mas, pela hipótese de indução, sabemos que $P(k)$ é verdade, ou seja, para qualquer grafo G com k arestas a fórmula de Euler é válida. Então, como G' tem k arestas, podemos escrever:

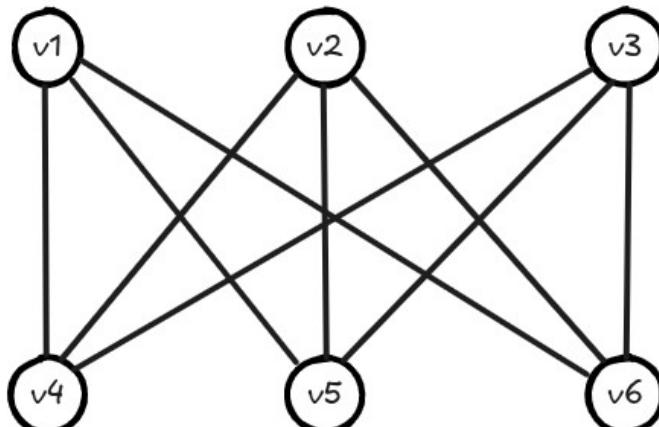
$$n' - m' + f' = 2$$

5. Como ao passar de G para G' eliminamos um ciclo, temos $f' = f - 1$ e

$$n - m + f = n' - (m' + 1) + (f' + 1) = n' - m' + f' = 2 \quad (\text{pelo passo anterior}).$$

Portanto, a prova está concluída.

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



$K_{3,3}$

Prova (por contradição):

1. Suponha inicialmente que o grafo $K_{3,3}$ seja representado por um grafo plano.
2. Sabemos que $n = 6$ e $m = 9$, porém precisamos elaborar em relação ao número de faces.
3. Como $K_{3,3}$ é bipartido, o comprimento do menor ciclo é 4 (não tem ciclo ímpar).
4. Assim, todas as faces ou regiões devem ter pelo menos 4 arestas.
5. Suponha que tenhamos k faces ou regiões: R_1, R_2, \dots, R_k e que $B(R)$ denote o número de arestas na borda da região R .
6. Seja $N = \sum_{i=1}^k B(R_i)$. Como cada região (face) é formada por pelo menos 4 arestas, então:

$$N \geq 4f$$

7. Entretanto, como cada aresta pode ser contada até duas vezes (quando está na fronteira entre 2 regiões), temos:

$$N \leq 2m = 18$$

8. Assim, podemos escrever $4f \leq 18$, o que implica em $f \leq \frac{9}{2} = 4.5$

9. Mas, usando a fórmula de Euler, chega-se em:

$$n - m + f = 6 - 9 + f = -3 + f$$

10. Como $f \leq 4.5$, temos que $-3 + f \leq 1.5$, o que é uma contradição, pois de acordo com a fórmula de Euler, esse valor deveria ser igual a 2.

Portanto, o grafo bipartido completo $K_{3,3}$ não é planar.

O problema do compartilhamento de recursos (The 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)

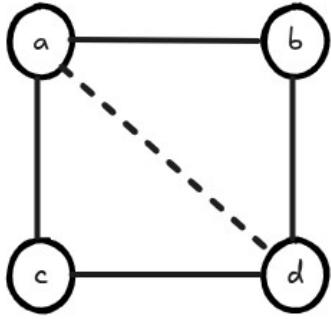
Problema: Dado um grafo G , como decidir se ele é planar ou não?

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?

Em outras palavras, podemos encontrar uma imersão planar para o $K_{3,3}$ no toro! Na prática, você pode desenhar um grafo $K_{3,3}$ no toro sem cruzar nenhuma aresta! Acredite, isso é possível!

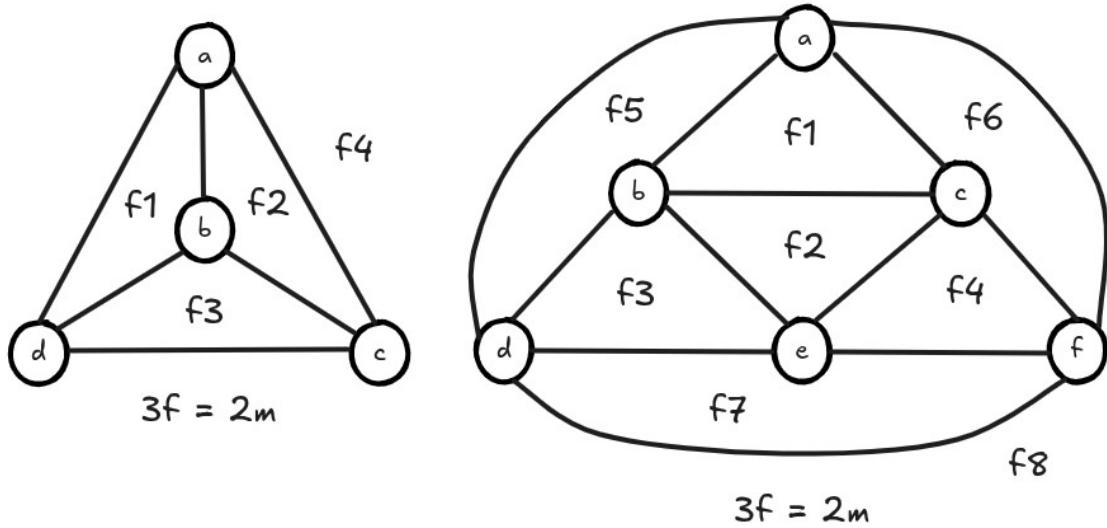
Teorema: Se G é planar com $|V| = n > 2$ e $|E| = m$ então $m \leq 3(n - 2)$

Represente G como um grafo plano (imersão planar). Pela fórmula de Euler $n - m + f = 2$. Toda face de G deve ser um triângulo pois se m é o número máximo de arestas, ele não pode ser aumentado dividindo uma face com uma aresta.



Com uma face quadrada,
o número de faces de G
pode ser aumentado

Nesse caso, como cada face é composta por 3 arestas, ao computar $3f$ estamos na verdade calculando 2 vezes o número de arestas, ou seja, $3f = 2m$. Note que contamos duas vezes cada aresta.



Assim, pela fórmula de Euler temos

$$n - m + \frac{2}{3}m = 2 \rightarrow n - \frac{1}{3}m = 2 \rightarrow \frac{3n - m}{3} = 2 \rightarrow 3n - m = 6 \rightarrow m = 3n - 6$$

Teorema: Todo grafo simples planar $G = (V, E)$ possui um vértice de grau no máximo 5.

Iremos provar por contradição. Sendo assim, assuma que todos os vértices de G tenham grau maior que 5, ou seja: $\forall v \in V (d(v) = 6)$. Então, pelo Handshaking Lema:

$$\sum_{i=1}^n d(v_i) = 2m \rightarrow 6n = 2m \rightarrow m = 3n$$

Pelo teorema anterior, se G é planar $m \leq 3n - 6$, o que implica em $3n \leq 3n - 6$, que é claramente uma contradição.

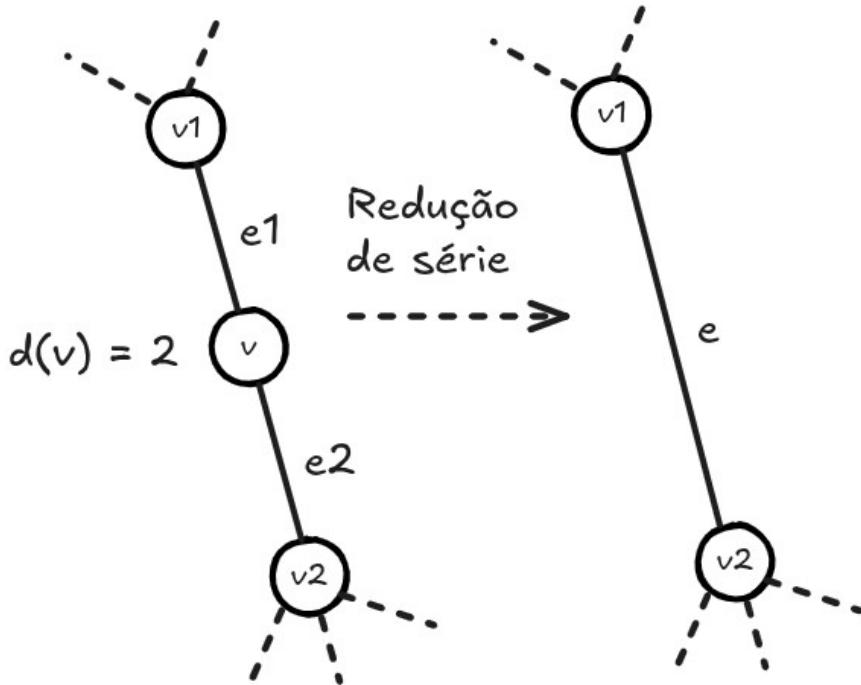
Obs: Resultados interessantes mas não podem ser usadas para decidir se G é planar. (para ser aplicado o grafo precisa estar desenhado como plano, então já preciso saber que é planar)

Veremos a seguir algumas definições importantes para a apresentação do resultado fundamental para a detecção de planaridade: o Teorema de Kuratowski.

Vimos que ambos K_5 e $K_{3,3}$ não são planares. Então, todo grafo que os contenha como subgrafo não pode ser planar. Mas, existem grafos que não contém K_5 ou $K_{3,3}$ como subgrafo e mesmo assim não são planares. Mas, tais grafos podem ser reduzidos a K_5 ou $K_{3,3}$ pela operação conhecida como redução de série.

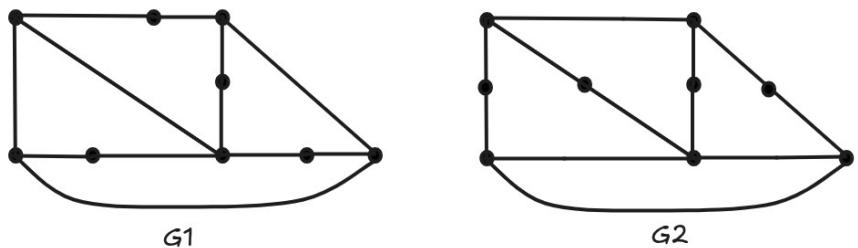
Def: Redução de série

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

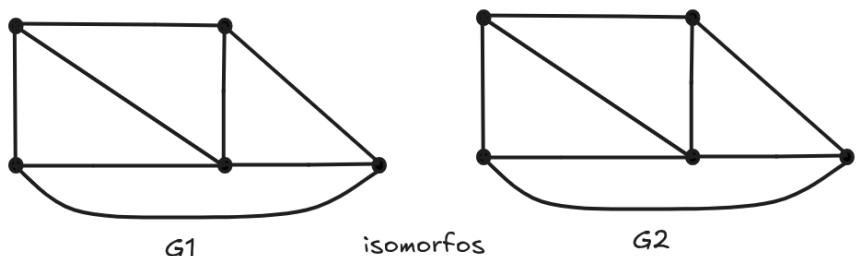


Def: Dois grafos G_1 e G_2 são homeomorfos se podem ser reduzidos a grafos isomorfos a partir de reduções de série.

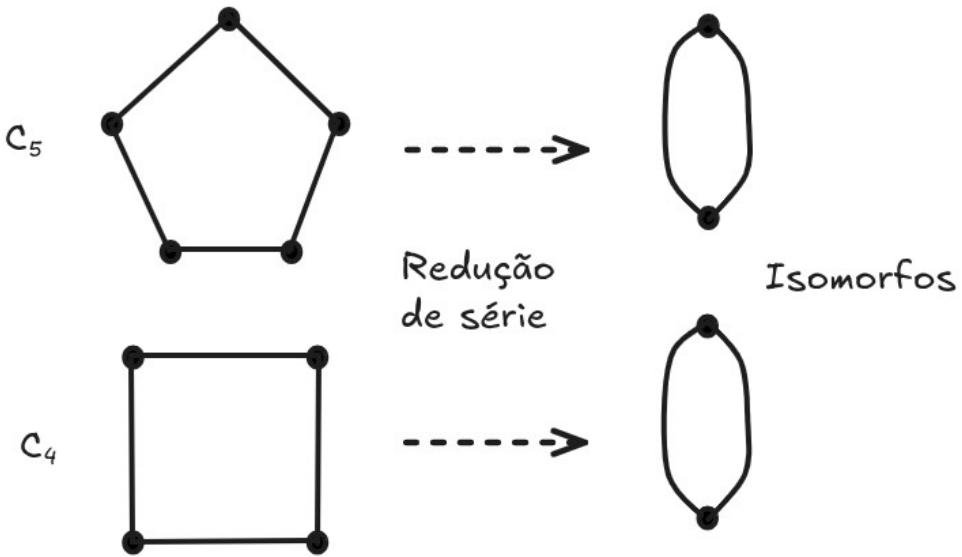
Por exemplo, os dois grafos a seguir são homeomorfos.



Isso pode ser facilmente percebido uma vez que ao aplicarmos reduções de séries em ambos, obtemos o seguinte resultado, ou seja, H_1 é isomorfo a H_2



Outro exemplo de grafos homeomorfos são os grafos ciclos. A figura a seguir ilustra que os grafos C_5 e C_4 são homeomorfos.



Problema (“Gênio indomável”): Liste todas as árvores homeomorfas irredutíveis de 10 vértices.
 Link: https://www.youtube.com/watch?v=iW_LkYiuTKE

Teorema de Kuratowski

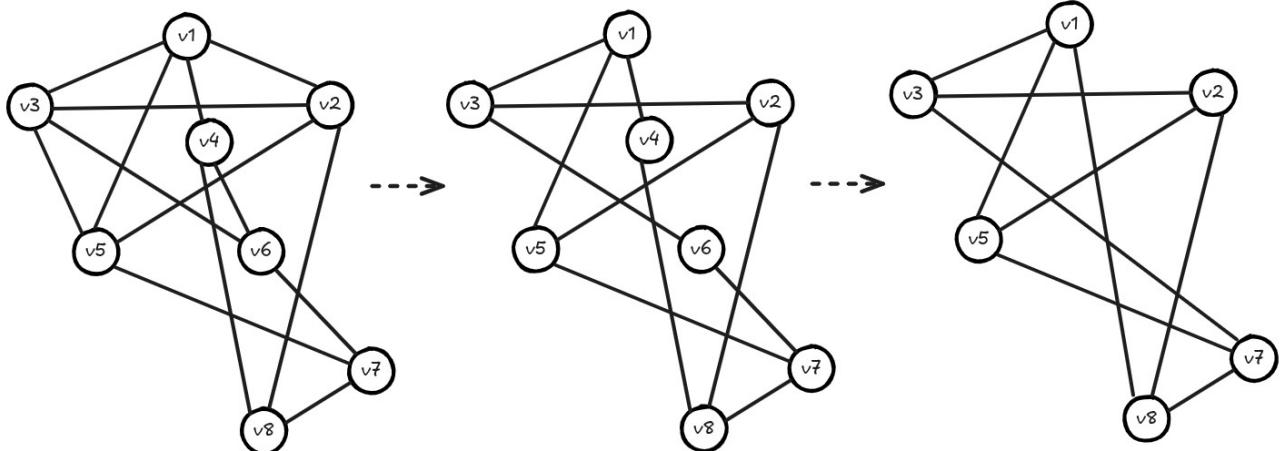
Um grafo G é planar se e somente se G não tiver um subgrafo homeomorfo a K_5 ou $K_{3,3}$

Critério objetivo, porém de difícil aplicação prática (algoritmos).

Em resumo esse resultado descreve como determinar se G é planar ou não através de 3 operações básicas:

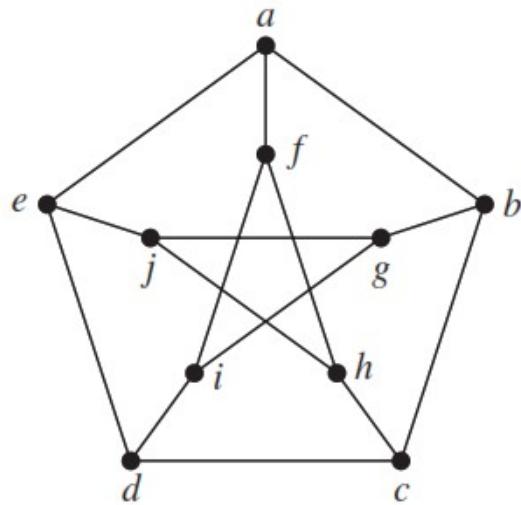
- i) remoção de arestas
- ii) remoção de vértices
- iii) redução de séries

Veremos a seguir um exemplo ilustrativo.



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 $K_{3,3}$, portanto G não é planar.

Ex: Mostre que o grafo de Petersen não é planar, utilizando o teorema de Kuratowski.



Planarity testing - testar a planaridade de grafos: atualmente existem algoritmos polinomiais.
Método pioneiro: 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

Coloração de vértices

Ideia: dado um grafo $G = (V, E)$, partitionar o conjunto de vértices V em conjuntos independentes.

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: partitionar 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 $k > 2$)? Trata-se de um problema NP-Completo

Em outras palavras queremos responder se dado G é possível encontrar 2, 3, 4, etc. conjuntos independentes.

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).

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 $\chi(G) \geq m$

Teorema: G é bipartido $\Leftrightarrow \chi(G) = 2$

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

Prova:

1. Inicie colorindo os vértices em uma sequência arbitrária.

2. Um vértice deve sempre receber uma cor livre, ou seja, diferente da cor dos vizinhos.

3. Seja o grau máximo de um vértice v em G , $\Delta(G)$.

4. Pelo princípio da casa dos pombos, sempre irá existir uma cor livre no conjunto $\{1, 2, 3, \dots, \Delta(G) + 1\}$.

Teorema: Seja $G = (V, E)$ um grafo conexo com $\Delta(G) \geq 3$. Se G não é completo, $\chi(G) \leq \Delta(G)$

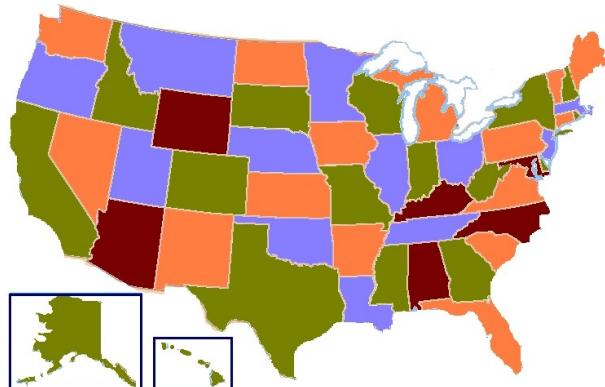
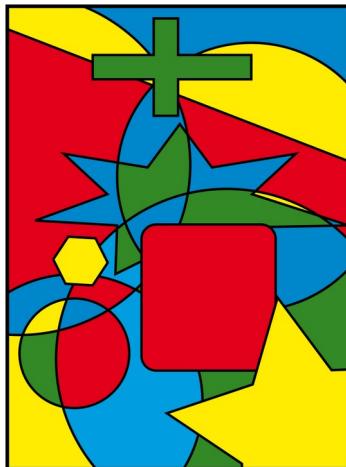
Teorema das 4 cores: Todo mapa pode ser colorido com apenas 4 cores, de modo que duas regiões vizinhas não recebam a mesma cor.

Em termos de grafos: O dual de um grafo básico simples planar possui $\chi(G) \leq 4$

- Problema que permaneceu em aberto por cerca de 125 anos

- Prova do teorema na década de 70: primeiro resultado teórico que contou com a assistência de um computador para enumerar um conjunto de possibilidades.

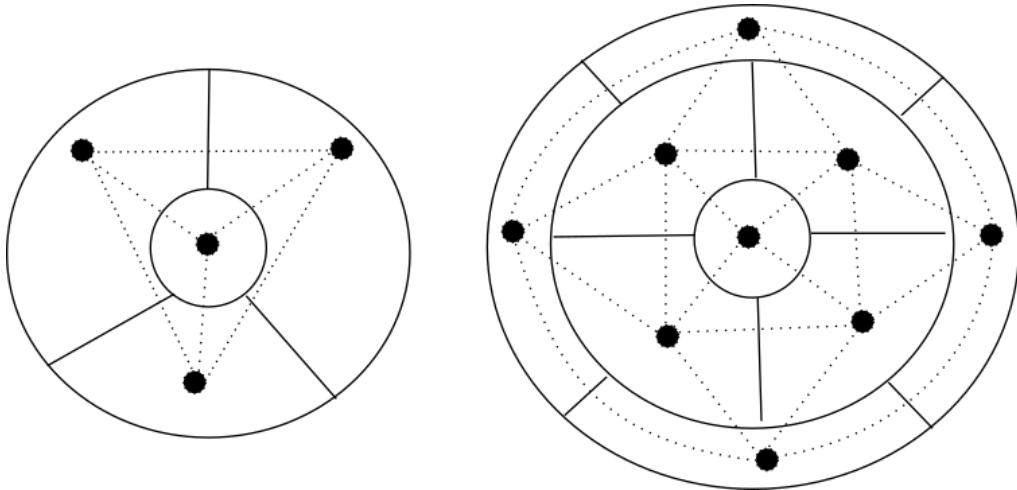
Desafio: produzir um mapa que necessite de mais do que 4 cores (invalidaria o teorema)



Def: Grafo dual

Seja $G = (V, E)$ um grafo básico simples planar. Um grafo dual G' de G é um grafo que tem um vértice para cada região (face) de G , e uma aresta para cada aresta em G que une duas regiões

adjacentes (teve haver uma aresta entre as fronteiras de G).



O centro de cada face do mapa torna-se um vértice e se 2 faces compartilham uma fronteira, há uma aresta entre elas. Todo mapa se transforma num grafo planar, pois mapas são estruturas planares. Um vídeo muito bom sobre o assunto pode ser assistido em no canal Numberphile do Youtube:
<https://www.youtube.com/watch?v=NgbK43jB4rQ>

Do ponto de vista da classificação dos problemas em computação, encontrar o número cromático $\chi(G)$ de grafos arbitrários é um problema NP-Hard. Alguns fatos relevantes:

1. Não se conhece algoritmo polinomial que colore um grafo arbitrário com exatamente $\chi(G)$ cores.
2. Existe um algoritmo polinomial para coloração de grafos que colore qualquer grafo G com no máximo $O\left(\frac{n}{\log n}\right)\chi(G)$

Algoritmos para coloração de vértices

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

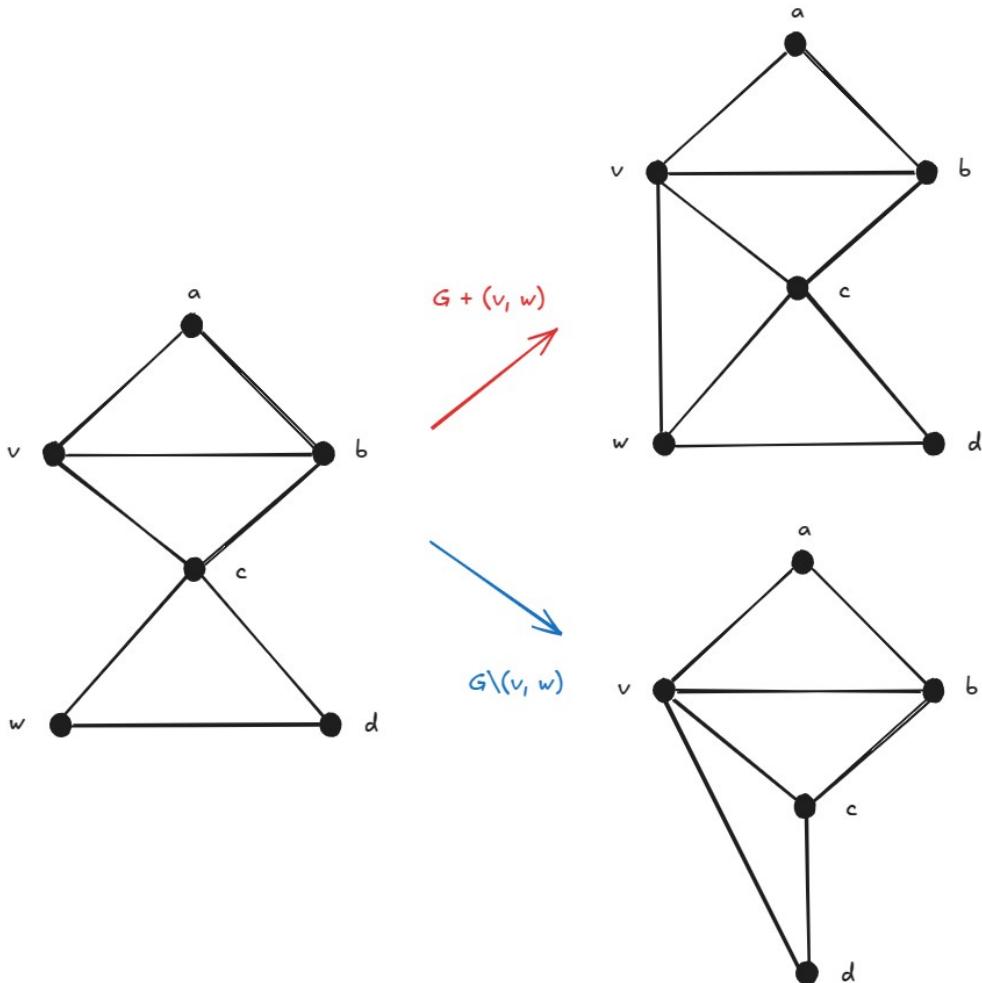
Problema NP-completo: não há garantias de que sempre retornem o verdadeiro $\chi(G)$

A seguir veremos como podemos definir um algoritmo exato para encontrar o número cromático de um grafo G.

Seja $G = (V, E)$ o grafo de entrada. Se $G = K_n$ então $\chi(G) = n$. Caso contrário, $\exists v, w \in V$ não adjacentes. A ideia é realizar duas alterações estruturais em G:

1. $\alpha_{v,w}(G) = G + (v, w)$ - adição da aresta (v, w) em G
2. $\beta_{v,w}(G) = G \setminus (v, w)$ - condensação dos vértices v e w

onde $G \setminus (v, w)$ é o grafo resultante da fusão dos vértices v e w e posterior eliminação de loops e arestas paralelas que venham a surgir. A figura a seguir ilustra um exemplo.



O método exato de coloração de vértices inicia verificando se $G = K_n$. Em caso positivo, $\chi(G)=n$. Caso contrário, determina-se $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$. O número cromático $\chi(G)$ será o mínimo entre os números cromáticos de $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$, que devem ser calculados recursivamente. Através da recursão, chegaremos em um grafo completo, onde $\chi(G)$ é facilmente computado (caso base). O resultado a seguir fundamenta o algoritmo exato em questão.

Teorema: Seja $G = (V, E)$ um grafo não completo e $v, w \in V$ um par de vértices não adjacentes. Então, o número cromático de G satisfaz:

$$\chi(G)=\min\{\chi(\alpha_{v,w}(G)), \chi(\beta_{v,w}(G))\}$$

Prova:

1. Suponha uma coloração ótima C de G .
2. Se os vértices v, w possuem cores distintas em C , então a aresta (v, w) pode ser adicionada em G sem alterar C . Neste caso, $\chi(G)=\chi(\alpha_{v,w}(G))$.
3. Se os vértices v, w possuem cores iguais a $c_i \in C$, então, ao fundirmos v, w em um único vértice z e atribuirmos a cor c_i a z mantendo as demais cores, temos $\chi(G)=\chi(\beta_{v,w}(G))$.
4. Portanto, $\chi(G)$ deve ser o menor valor entre $\chi(\alpha_{v,w}(G))$ e $\chi(\beta_{v,w}(G))$.

Note que $\chi(G)$ é igual ao número de vértices do menor grafo completo obtido através das operações $\alpha_{v,w}(G)$ e $\beta_{v,w}(G)$.

O método a seguir define um algoritmo exato para o problema da coloração de vértices.

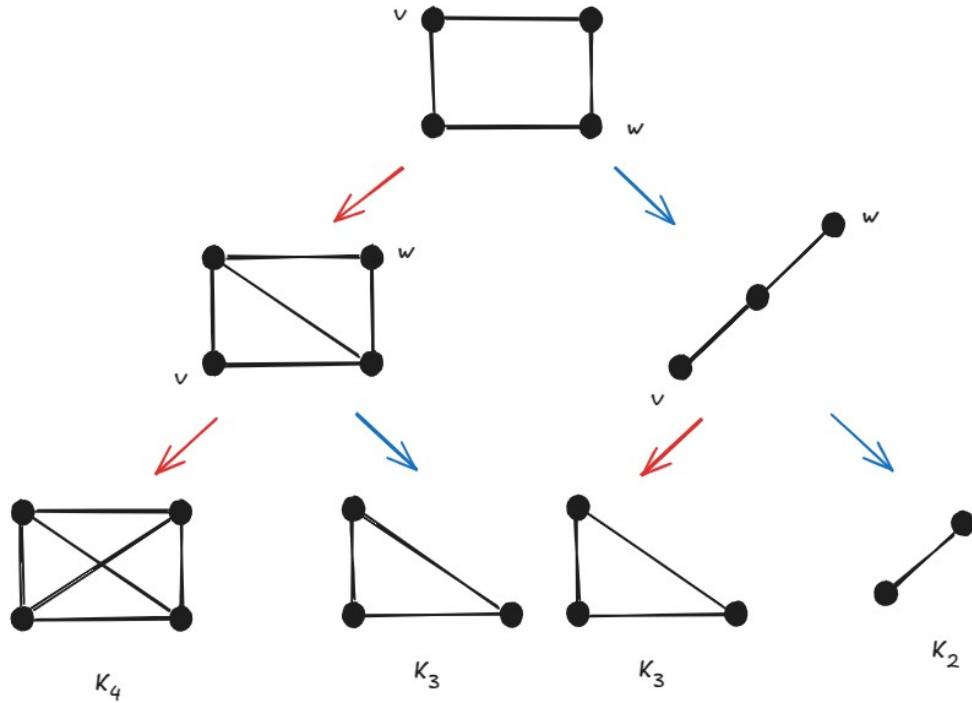
```

Chromatic_Number(G) {
    n = |V|
    if G == Kn
        χ = min{χ, n}
    else {
        Encontre um par de vértices não adjacentes v, w em G
        Chromatic_Number(αv,w(G))
        Chromatic_Number(βv,w(G))
    }
}

χ = n
Chromatic_Number(G)

```

O algoritmo acima dispara um processo recursivo que constrói uma árvore binária em que cada nó corresponde a um grafo: se G é um nó corrente, seu filho a esquerda é $\alpha_{v,w}(G)$ e seu filho a direita é $\beta_{v,w}(G)$. As folhas da árvore, que indicam a condição de parada da recursão, serão sempre grafos completos. A figura a seguir ilustra um exemplo simples.



Portanto, o número cromático de G é $\chi(G)=2$.

Análise da complexidade

Note que o número máximo de grafos a serem examinados é igual ao número de nós total na árvore binária. Como em cada nível ligamos/fundimos apenas 2 vértices, a profundidade da árvore é proporcional ao número de arestas no grafo complementar de G , denotado por m' .

$$2^0 + 2^1 + 2^2 + \dots + 2^{m'} = \sum_{i=0}^{m'} 2^i$$

Note que $2^{i+1} - 2^i = 2^i + 2^i$, o que nos leva a $2^i = 2^{i+1} - 2^i$. Dessa forma temos:

$$\sum_{i=0}^{m'} (2^{i+1} - 2^i) = 2^{m'+1} - 2^0 = 2^{m'+1} - 1$$

ou seja, a complexidade é exponencial, $O(2^{m'})$. Portanto, esse algoritmo, torna-se inviável para grafos com muito vértices. A seguir iremos discutir um algoritmo aproximado que utiliza uma estratégia gulosa para encontrar o número cromático de G .

Do ponto de vista da classificação dos problemas em computação, encontrar o número cromático $\chi(G)$ de grafos arbitrários é um problema NP-Hard. Alguns fatos relevantes:

1. Não se conhece algoritmo polinomial que colore um grafo arbitrário com exatamente $\chi(G)$ cores.
2. Existe um algoritmo polinomial para coloração de grafos que colore qualquer grafo G com no máximo $O\left(\frac{n}{\log n}\right)\chi(G)$ cores (um pouco acima do valor mínimo).

Abordagem gulosa para coloração de vértices

Suponha que os vértices estejam ordenados em ordem decrescente dos graus, ou seja, temos v_1, v_2, \dots, v_n e seus respectivos graus $\Delta = d_1 \geq d_2 \geq \dots \geq d_n$

Teorema: $\chi(G) = \max_{1 \leq i \leq n} \min\{d_i + 1, i\}$

Prova:

1. No instante em que colorimos v_i (i -ésimo vértice da sequência), note que:

a) No máximo $i - 1$ cores foram utilizadas em seus vizinhos, pois antes dele foram exatamente $i - 1$ vértices.

b) Mas também, no máximo d_i cores são utilizadas pelos vizinhos, uma vez que o grau de v_i é justamente d_i

Esse resultado fundamenta o algoritmo de Welsh & Powell, utilizado para colorir um grafo e aproximar o verdadeiro valor do número cromático de G , e que será apresentado a seguir.

```
Welsh_Powell(G) {
    Ordene os vértices em ordem decrescente de graus
    for each  $v_i \in V$ 
         $C_i = [1, 2, \dots, i]$  # Lista das cores do vértice  $v_i$ 
    for  $i = 1$  to  $n$  {
        color =  $C_i[1]$  # Escolhe a primeira cor da lista
         $v_i.c = color$  # Colore  $v_i$  com a cor escolhida
```

```

        for each  $v_j$  in  $N(v_i)$ 
            # Remove a cor da lista dos vizinhos
             $C_j = C_j - \text{color}$ 
    }
     $\chi = 0$ 
    for each  $v_i \in V$  {
        if  $v_i.c > \chi$ 
             $\chi = v_i.c$ 
    }
    return  $\chi$ 
}

```

Análise da complexidade

1. Note que a ordenação dos vértices pode ser realizada em $O(n \log n)$

2. A criação das listas de cores envolve trabalhar com n listas de tamanhos variáveis. Para o i -ésimo vértice, a lista de cores tem tamanho i , de modo que o número de operações é dado por:

$$1+2+3+4+\dots+n=\sum_{i=1}^n i$$

3. Mas é possível reduzir esse custo, sabendo que o número cromático em um grafo que não é o K_n é limitado superiormente pelo maior grau. Seja K o maior grau de um vértice em G , então basta que as listas de cores contenham K cores distintas e não n , ou seja, o número de operações gastas é:

$$1+2+3+\dots+K+K+K \leq nK$$

Portanto, a etapa de criação das listas de cores tem complexidade $O(nK)$

4. O loop principal do algoritmo percorre todo o vértice v de G uma vez e para cada vértice, acessamos os seus $d(v)$ vizinhos. Se utilizarmos um array estático, podemos apagar a cor color da lista C_i em $O(1)$. Logo, temos que a complexidade desse loop é:

$$d(v_1)+d(v_2)+\dots+d(v_n)=2m \text{ ,ou seja, } O(m) .$$

5. Por fim, o ultimo loop serve apenas para encontrar o maior rótulo utilizado como cor, o que é equivalente a encontrar o máximo do vetor. Assim, o custo é $O(n)$.

Portanto, o custo total é: $O(n \log n)+O(nK)+O(m)+O(n)$. Note que no pior caso, $m=O(n^2)$ o que nos leva a conclusão de que a complexidade do algoritmo Welsh and Powell é $O(n^2)$.

A seguir apresentaremos alguns exemplos de problemas que podem ser resolvidos através da coloração de vértices.

O Problema da Alocação de Frequências

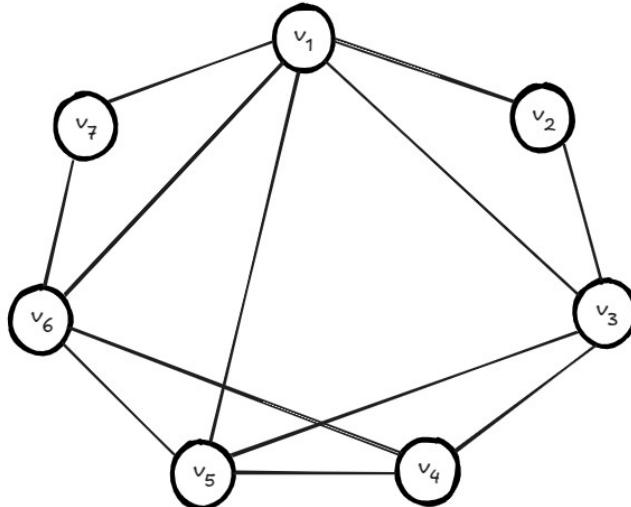
Considere 7 antenas de transmissão de sinais de telefonia. Sabe-se que devido a fatores como localização geográfica e tipo de serviço oferecido, as antenas possuem regiões de influência de modo que tem-se a seguinte matriz de interferências:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1		x	x		x	x	x
v_2	x		x				
v_3	x	x		x	x		
v_4			x		x	x	
v_5	x		x	x		x	
v_6	x			x	x		x
v_7	x					x	

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

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

1º passo: Devemos construir o grafo de incompatibilidade. Neste grafo, dois vértices devem ser ligados por uma aresta se existe interferência entre as duas antenas.



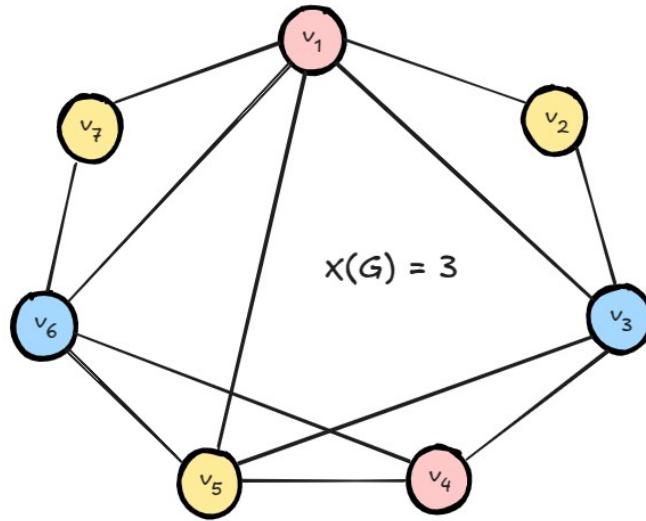
2º passo: Ordenar os vértices em ordem decrescente de graus: $v_1, v_3, v_5, v_6, v_4, v_2, v_7$

3º passo: Definir as listas de cores para cada vértice

4º passo: Colorir os vértices iterativamente removendo as cores usadas das listas dos vizinhos.

	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}
$v_1 \leftarrow 1$	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}
$v_3 \leftarrow 2$		x	C5={3}	C6={2,3,4}	C4={1,3,4,5}	C2={3,4,5}	C7={2,3,4,5}
$v_5 \leftarrow 3$			x	C6={2,4}	C4={1,4,5}	C2={3,4,5}	C7={2,3,4,5}
$v_6 \leftarrow 2$				x	C4={1,4,5}	C2={3,4,5}	C7={3,4,5}

$v4 \leftarrow 1$				x	$C2 = \{3, 4, 5\}$	$C7 = \{3, 4, 5\}$
$v2 \leftarrow 3$				x	$C7 = \{3, 4, 5\}$	
$v7 \leftarrow 3$				x		



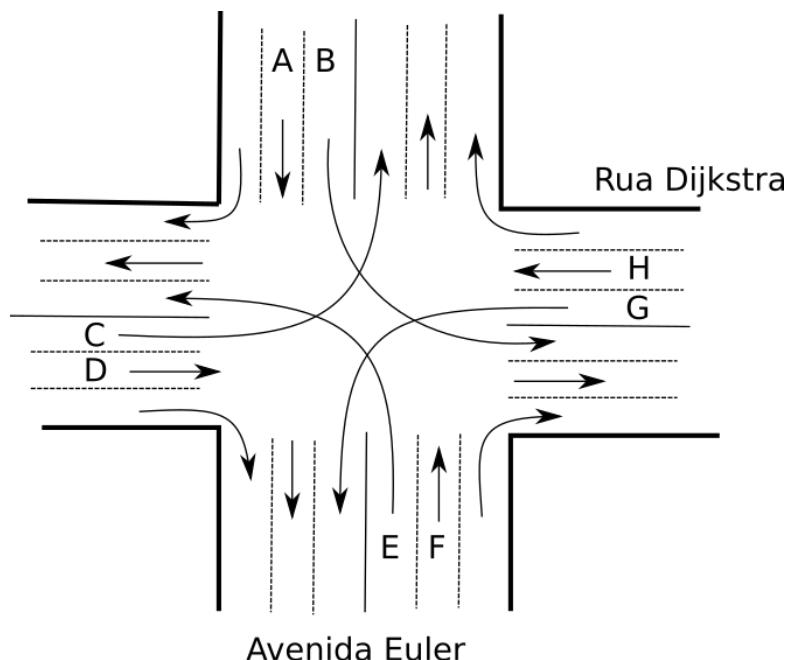
A aproximação para o número cromático obtida pelo algoritmo guloso é 3, o que, neste caso, corresponde ao verdadeiro valor de $\chi(G)$. As partições obtidas são:

$$\begin{aligned} P_1 &= \{v_1, v_4\} \\ P_2 &= \{v_3, v_6\} \\ P_3 &= \{v_2, v_5, v_7\} \end{aligned}$$

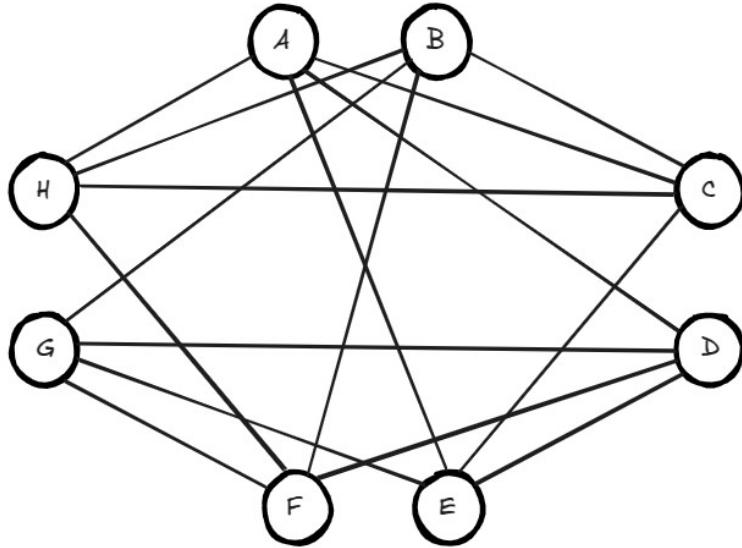
O problema do semáforo

O objetivo deste problema consiste em sincronizar os semáforos de um cruzamento utilizando coloração de vértices.

Dado o cruzamento de ruas a seguir, desenvolver um semáforo com o menor número de tempos possíveis. Qual deve ser o modo de funcionamento do mesmo.



1º passo: Consiste em gerar o grafo de incompatibilidade. Para isso cada uma das faixas de A a H deve ser um vértice e a cada possível cruzamento de faixas deve existir uma aresta (ou seja, deve existir uma aresta sempre que for possível ocorrer uma batida). O grafo resultante é ilustrado pela figura a seguir.



2º passo: Consiste na aplicação do algoritmo guloso (Welsh & Powell) para colorir o grafo acima utilizando o menor número de cores possíveis. Como o grau máximo é 4 e o grafo não é completo, temos garantia de que o número cromático é menor ou igual a 4. A execução desse passo é deixada como exercício para o leitor.

O problema do aeroporto

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.

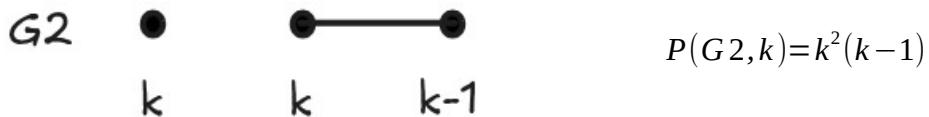
Polinômio cromático

Def: Para um grafo $G = (V, E)$, seja $P(G, k)$ o número de k -colorações válidas de G . Pode-se mostrar que $P(G, k)$ é um polinômio para todo grafo G .

Iremos iniciar com grafos simples e calcular os polinômios cromáticos fundamentais. Primeiro, iremos iniciar com um grafo nulo de 3 vértices. Note que nesse caso, podemos utilizar k cores em cada vértice, pois como não há arestas, nenhum vértice é vizinho de outro.



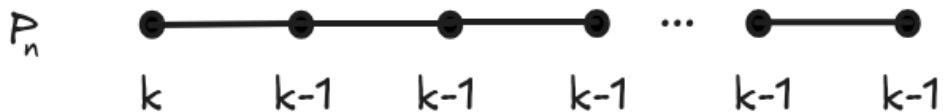
Note que ao adicionarmos uma aresta no grafo anterior, temos:



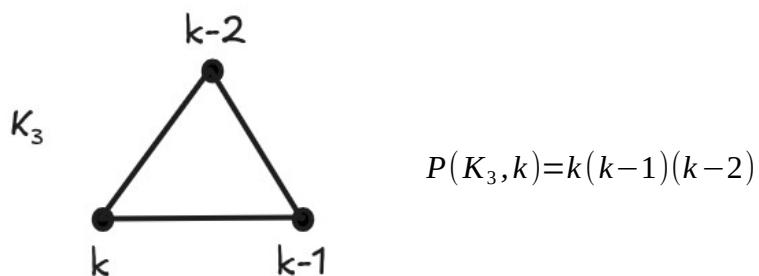
pois a cor do segundo vértice não pode ser utilizada para colorir o terceiro. Ao adicionarmos mais uma aresta, o polinômio cromático fica:



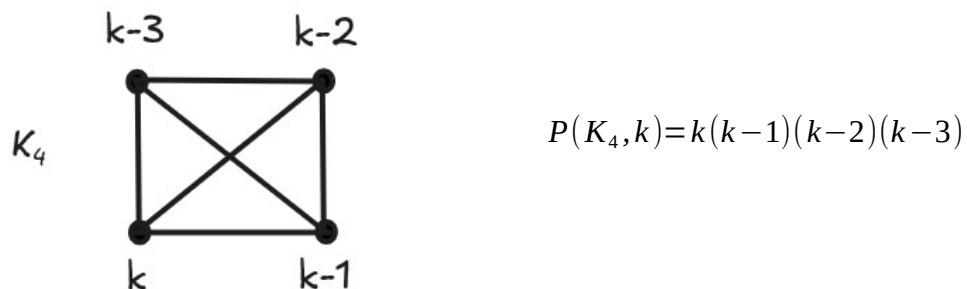
É fácil notar que o grafo caminho de n vértices, conhecido como P_n (path), possui o seguinte polinômio cromático: $P(P_n, k) = k(k-1)^{n-1}$



O grafo completo K_3 (triângulo) possui outro polinômio fundamental:



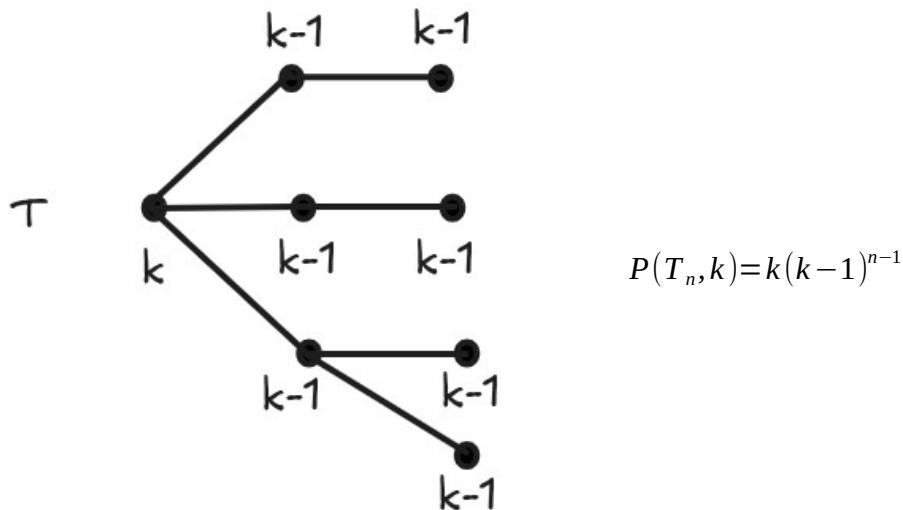
No caso do grafo completo de 4 vértices, K_4 , possui o seguinte polinômio cromático:



Generalizando para o grafo completo de n vértices:

$$K_n \quad P(K_n, k) = k(k-1)(k-2)\dots(k-(n-1))$$

No caso de árvores, note que o polinômio cromático é idêntico ao do grafo caminho. Aliás, isso já era esperado, uma vez que o grafo caminho é uma árvore.



Porém, uma pergunta que surge é: como calcular o polinômio cromático para um grafo arbitrário? O resultado a seguir é fundamental para responder a essa pergunta.

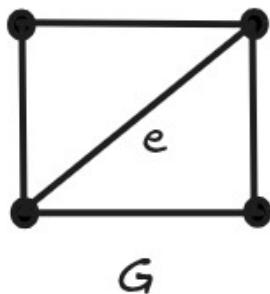
Teorema fundamental da redução

Seja $G = (V, E)$ um grafo básico simples e e uma aresta do conjunto E . Então:

$$P(G, k) = P(G - e, k) - P(G/e, k)$$

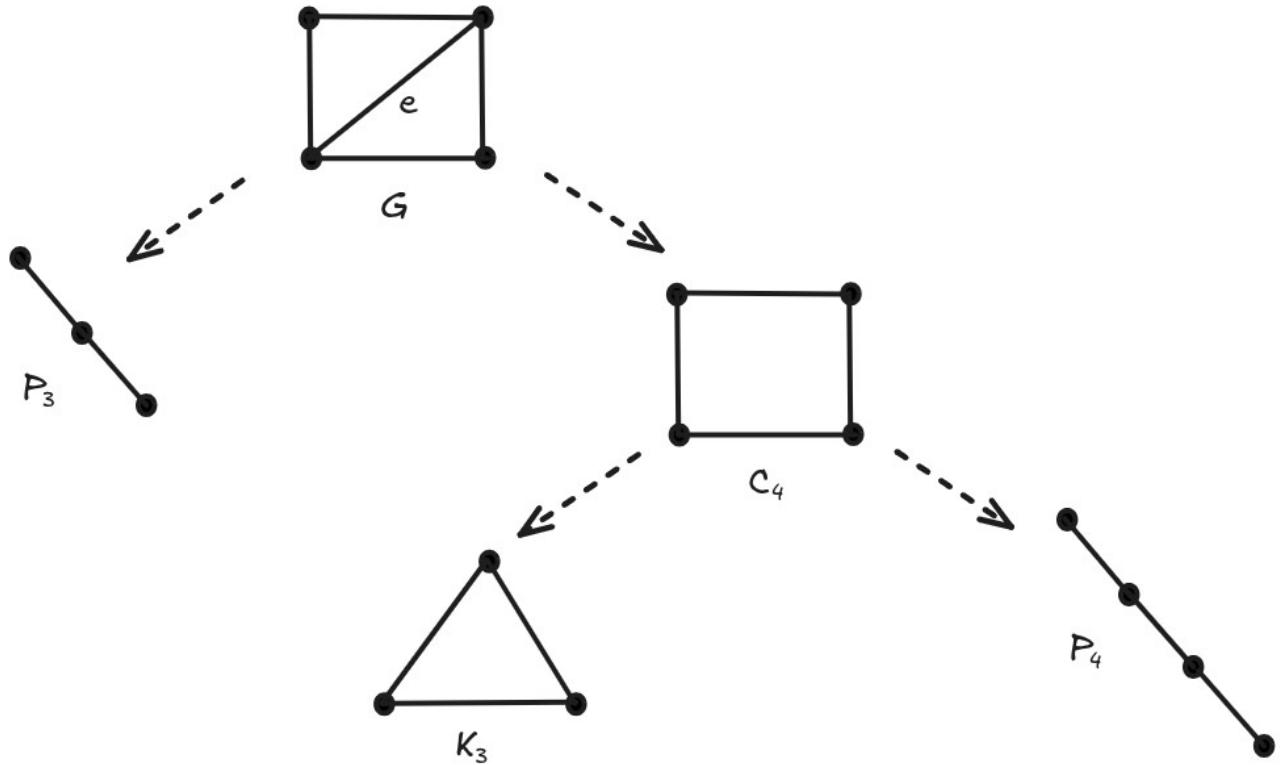
onde G/e denota o grafo obtido a partir da contração da aresta e . Devido a complexidade matemática, iremos omitir a prova desse resultado, mas a seguir iremos aplicá-lo em um exemplo prático.

Ex: Determine o polinômio cromático de G a seguir:



Iremos aplicar o teorema fundamental da redução para gerar a árvore de decomposição do grafo. No primeiro estágio, devemos decompor os níveis da árvore até chegar em um polinômio fundamental e em seguida, voltar na árvore calculando os polinômios cromáticos.

A árvore de decomposição fica:



Iniciamos pela aplicação do teorema fundamental da redução em G :

$$P(G, k) = P(G - e, k) - P(G/e, k) = P(C_4, k) - P(P_3, k)$$

Como P_3 tem um polinômio fundamental, é folha da árvore de decomposição. Assim, temos:

$$P(P_3, k) = k(k-1)^2$$

Esta folha está encerrada. Vamos agora aplicar o teorema da redução em C_4 :

$$P(C_4, k) = P(C_4 - e, k) - P(C_4/e, k) = P(P_4, k) - P(K_3, k)$$

Note que ambos P_4 e K_3 possuem polinômios fundamentais, o que nos leva a:

$$P(P_4, k) = k(k-1)^3$$

$$P(K_3, k) = k(k-1)(k-2)$$

Como ambos são folhas, podemos voltar na árvore e computar o polinômio de C_4 como:

$$P(C_4, k) = k(k-1)^3 - k(k-1)(k-2) = k(k-1)[(k-1)^2 - (k-2)]$$

Expandindo o quadrado, temos:

$$P(C_4, k) = k(k-1)[k^2 - 2k + 1 - k + 2] = k(k-1)(k^2 - 3k + 3)$$

Agora que temos o polinômio de C_4 , podemos voltar e calcular o polinômio final de G como:

$$P(G, k) = k(k-1)(k^2 - 3k + 3) - k(k-1)^2$$

Colocando o termo $k(k-1)$ em evidência:

$$P(G, k) = k(k-1)[(k^2 - 3k + 3) - (k-1)] = k(k-1)(k^2 - 4k + 4)$$

Fatorando o polinômio quadrático, finalmente chegamos em:

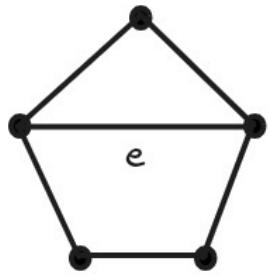
$$P(G, k) = k(k-1)(k-2)^2$$

Pergunta: quantas 5 colorações válidas existem para o grafo G em questão?

$$P(G, 5) = 5 \times 4 \times 3^2 = 20 \times 9 = 180$$

Exercício: Utilizando o teorema fundamental da redução, encontre o polinômio cromático dos grafos a seguir. Mostre a árvore de decomposição.

a)



b)

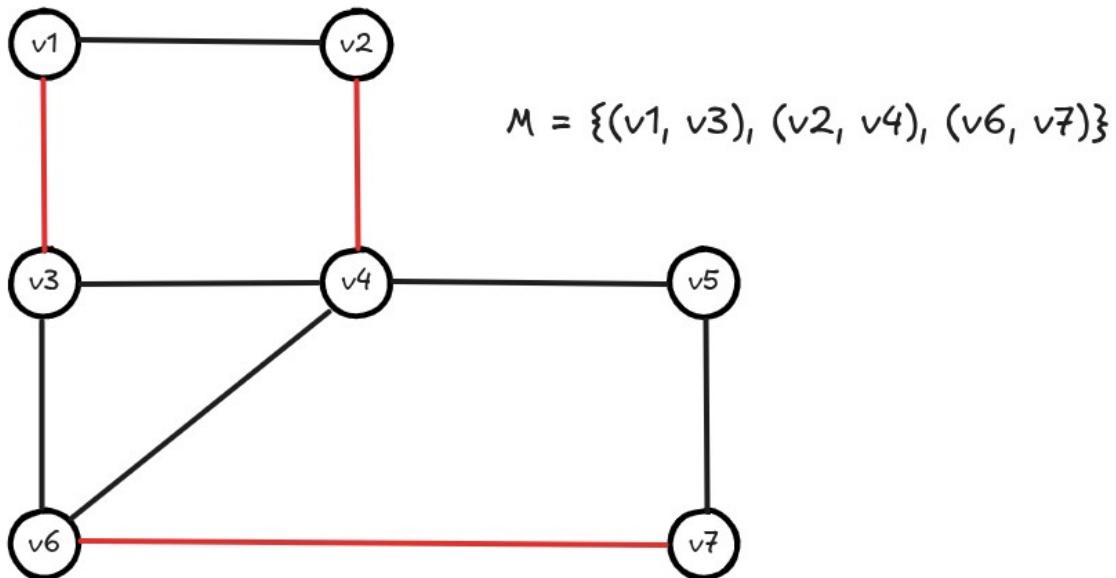


Emparelhamentos (Matchings)

Emparelhamentos em grafos são subconjuntos de arestas que conectam pares de vértices sem sobreposição, ou seja, nenhum vértice pertence a mais de uma aresta do emparelhamento. Esse conceito é fundamental na teoria dos grafos e tem ampla aplicação prática em problemas de alocação, emparelhamento de recursos, designação de tarefas e organização de preferências. Em grafos bipartidos, por exemplo, os emparelhamentos são utilizados para modelar situações como a atribuição de estudantes a projetos, empregos a candidatos ou órgãos a pacientes. A relevância dos emparelhamentos está não apenas na variedade de contextos em que podem ser aplicados, mas também na riqueza algorítmica envolvida, com soluções clássicas como o algoritmo húngaro para emparelhamento máximo com pesos e o algoritmo de Gale-Shapley para emparelhamentos estáveis. O estudo de emparelhamentos contribui significativamente para a compreensão de estruturas combinatórias e o desenvolvimento de algoritmos eficientes em otimização discreta.

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: v1, v2, v3, v4, v6, v7

Def: Vértice M-não-saturado

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

Ex: v5

Def: Emparelhamento perfeito

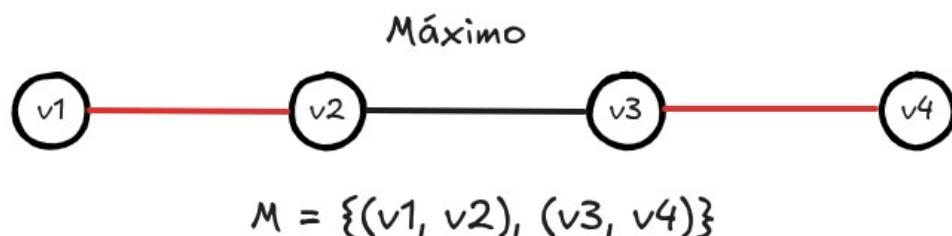
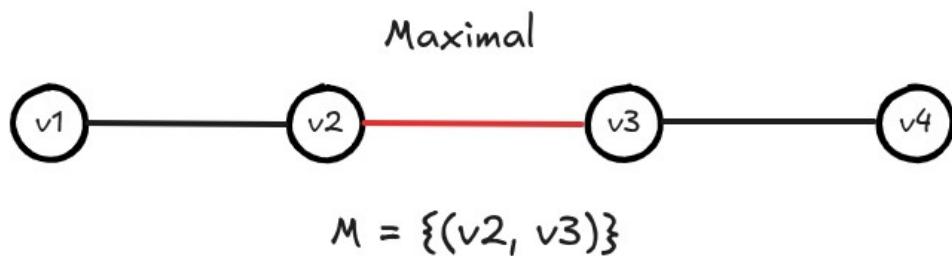
$\exists M \subset E$ tal que $\forall v \in V$ v é M-saturado $\rightarrow M$ é perfeito
(possível apenas se G tem um número par de vértices)

Def: Emparelhamento maximal

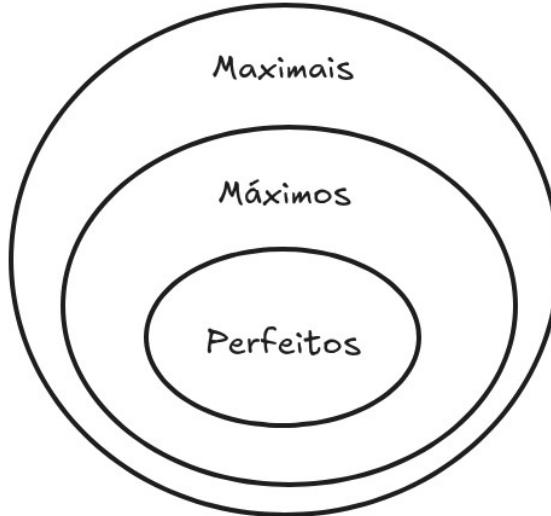
Se M não pode ser aumentado com o acréscimo de uma aresta então M é maximal

Def: Emparelhamento máximo

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



A figura a seguir ilustra a relação entre emparelhamentos perfeitos, máximos e maximais.



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

Teorema: Sejam M_1 e M_2 dois emparelhamentos distintos em G. Seja H o subgrafo definido pela diferença simétrica entre M_1 e M_2 :

$$M_1 \Delta M_2 = (M_1 - M_2) \cup (M_2 - M_1) \quad (\text{arestas que estão ou em } M_1 \text{ ou em } M_2 \text{ mas não em ambos})$$

Então, cada componente conexa de H é de um de dois tipos:

- a) um ciclo de comprimento par cujas arestas estão alternadamente em M_1 e M_2 ;
- b) um caminho cujas arestas estão alternadamente em M_1 e M_2 e cujos vértices extremidades são insaturados em um dos dois emparelhamentos;

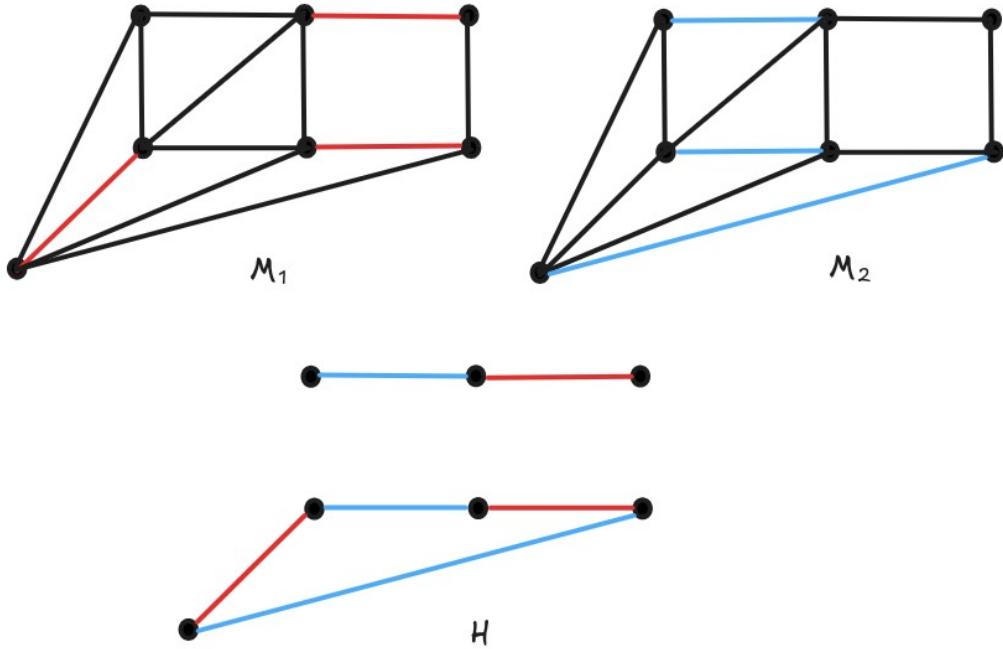
Prova:

Seja v um vértice arbitrário de H. Então, temos:

- i) v é um vértice extremidade de uma aresta em $M_1 - M_2$ e também de uma aresta em $M_2 - M_1$;
- ii) v é um vértice extremidade de uma aresta de $M_1 - M_2$ ou de $M_2 - M_1$, mas não de ambos;

Em ambos os casos i) e ii), como M_1 é um emparelhamento, existe no máximo uma aresta e em M_1 com o vértice v como sua extremidade. De modo similar, existe no máximo uma aresta em M_2 com o vértice v como extremidade. Assim, no caso i) v tem grau dois em H enquanto que no caso ii) v tem grau um em H. A figura a seguir ilustra o resultado desse teorema.

□



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 se e somente se G não possui caminho M -aumentado

1. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

M é máximo \rightarrow não há caminho M -aumentado = Há caminho M -aumentado $\rightarrow M$ não é máximo

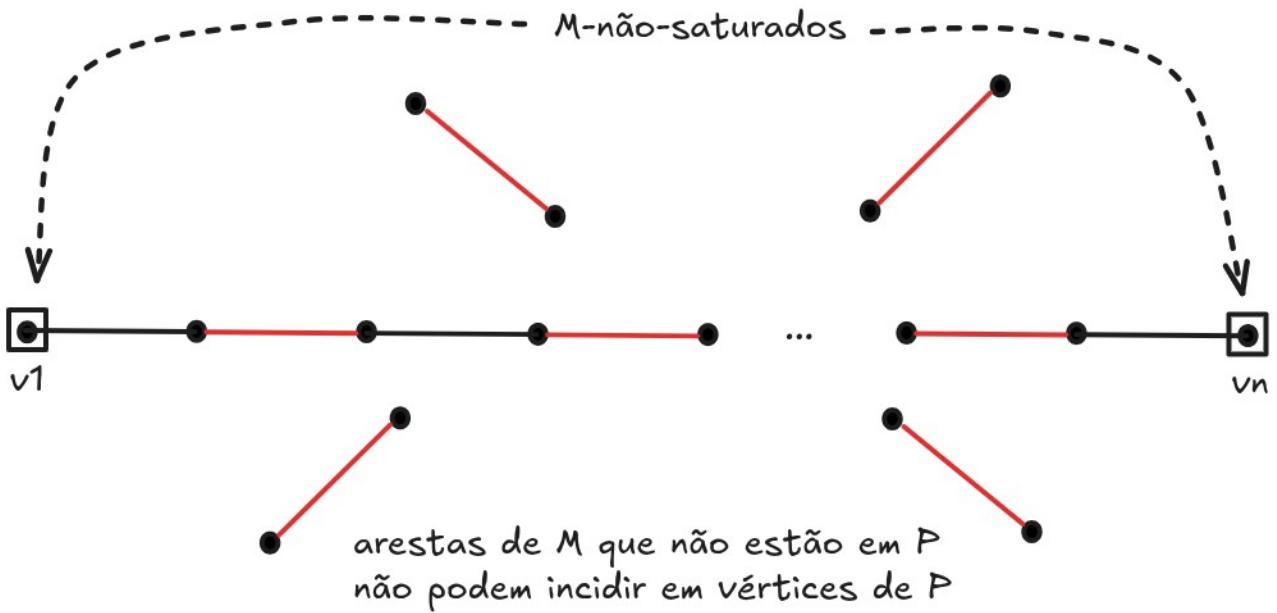
Seja um emparelhamento $M \subset E$. Então, há 2 tipos de arestas no grafo G :

- 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:

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

Pela alternância das arestas, temos que para cada clara devemos ter uma escura. Então se existem m 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 $2m + 1$. Note que é um número ímpar não importa o valor de n (há portanto uma aresta clara a mais que escura). É importante notar que em um caminho M -aumentado o número de arestas claras é sempre maior que o número de arestas escuras.



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 $|M'| = |M| + 1$. A operação que consiste em transformar M em M' usando P é chamada de **Transferência ao longo do caminho M -aumentado P** .

A prova da volta consiste em verificar que se não há caminho M -aumentado então M deve ser um emparelhamento máximo.

2. (volta) $q \rightarrow p = \neg q \rightarrow \neg p$

Não existe caminho M -aumentado $\rightarrow M$ é emparelhamento máximo

Seja M' um emparelhamento máximo arbitrário em G . Deseja-se mostrar que $|M| = |M'|$, de modo que se M tiver o mesmo número de arestas que um emparelhamento máximo, ele também será máximo.

Seja H o subgrafo definido pelo conjunto de arestas dado pela diferença simétrica entre M e M' :

$$M \Delta M' = (M - M') \cup (M' - M)$$

Pelo teorema anterior, sabemos que os componentes conexos de H são:

- a) ciclos de comprimento par com arestas alternadamente em M e M' ; ou
- b) caminhos alternados com arestas de M e M' cujas extremidades são não saturadas em M ou M' ;

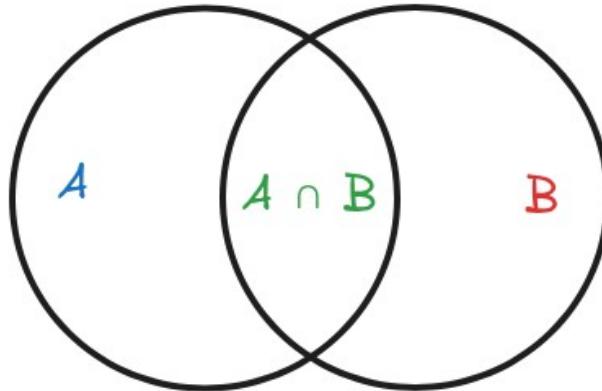
Se existe um caminho como o descrito em b) de comprimento ímpar, então tanto a origem quanto o destino são ambos não saturados em M ou ambos não saturados em M' , o que caracteriza um caminho M -aumentado (ou M' -aumentado).

Mas isso não pode ocorrer, pois de acordo com a hipótese inicial, não existe caminho M-aumentado e M' é máximo e portanto não admite caminho M' -aumentado.

Assim, os componentes de H são caminhos ou ciclos de comprimento par, de modo que contém o mesmo número de arestas de M e de M' . Matematicamente, isso implica em:

$$|M - M'| = |M' - M|$$

Da Teoria dos conjuntos, sabe-se que



$$|A| = |A - B| + |A \cap B|$$

de modo que podemos escrever:

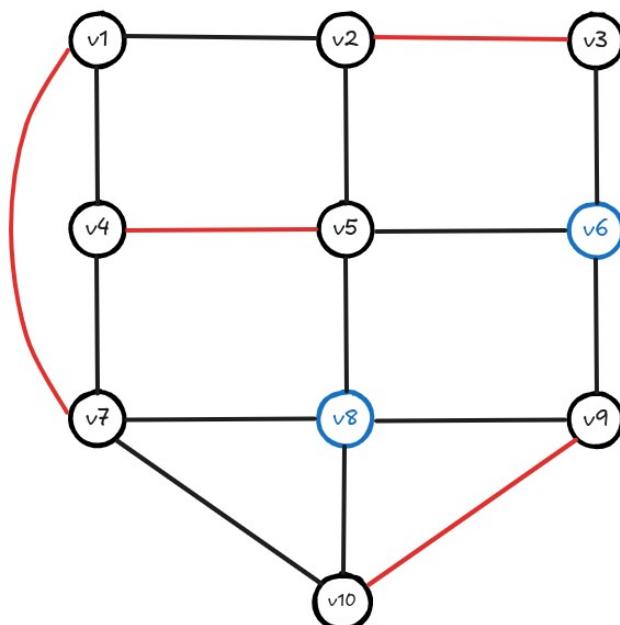
$$|M| = |M - M'| + |M \cap M'|$$

$$|M'| = |M' - M| + |M' \cap M|$$

Como $|M - M'| = |M' - M|$ segue que $|M| = |M'|$.

Portanto, o emparelhamento M é máximo, o que conclui a prova.

Ex:

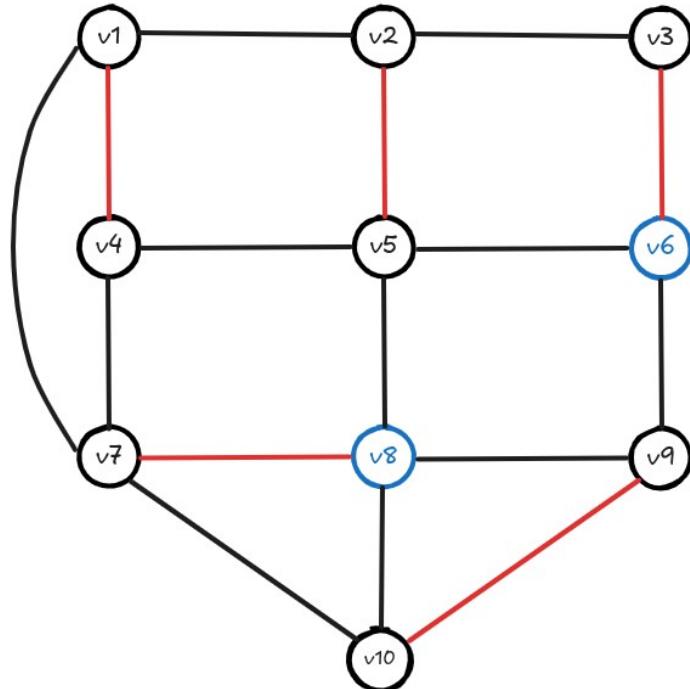


$M = \{(v1, v7), (v2, v3), (v4, v5), (v9, v10)\}$. M é máximo? Porque? Justifique

Note que existe o caminho M -aumentado $P = v6 \rightarrow v3 \rightarrow v2 \rightarrow v5 \rightarrow v4 \rightarrow v1 \rightarrow v7 \rightarrow v8$, 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:

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

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

$|N_G(S)| \geq |S|$ onde $N_G(S)$ denota o conjunto vizinhança de S no grafo G (todos elementos de G alcançáveis a partir dos elementos de S)

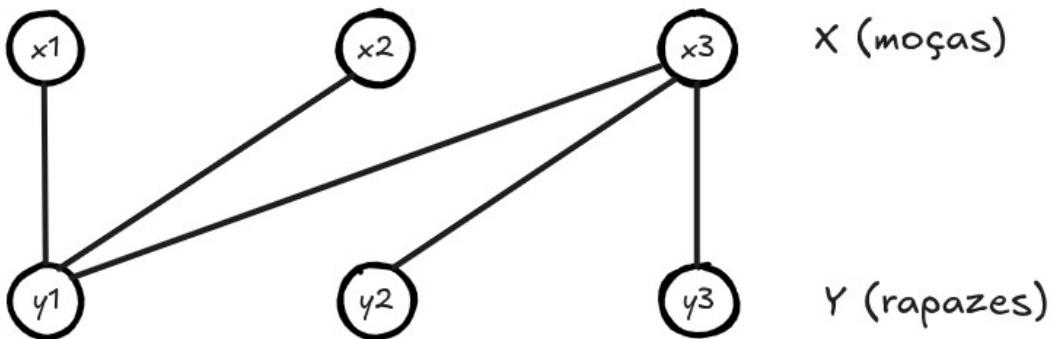
Prova:

A. (ida) $p \rightarrow q = \neg q \rightarrow \neg p$

$\exists M$ que satura todo x em $X \rightarrow \forall S \in X |S| \leq |N_G(S)|$

$\exists S \in X |S| > |N_G(S)| \rightarrow \nexists M$ que satura todo x em X

1. Vamos supor que X é o conjunto de garotas. Se existe um subconjunto de k garotas que coletivamente conhecem um conjunto de $m < k$ garotos, então é impossível casar todas elas, ou seja, não existe emparelhamento que satura todo x em X .



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 $O(2^n)$)

$$S_1 = \{x_1\}, N(S_1) = \{y_1\}$$

$$S_2 = \{x_2\}, N(S_2) = \{y_1\}$$

$$S_3 = \{x_3\}, N(S_3) = \{y_1, y_2, y_3\}$$

$$S_4 = \{x_1, x_2\}, N(S_4) = \{y_1\}$$
 (Falhou)

Como há duas garotas que conjuntamente só conhecem um único garoto, nesse grafo não será possível encontrar um emparelhamento M que sature todos os vértices de X (impossível).

B. (volta) $q \rightarrow p$ (prova por indução em X , mais complicada)

$\forall S \in X |S| \leq |N_G(S)| \rightarrow \text{Existe } M \text{ que satura todo } x \text{ em } X$

Caso base: $P(1)$: X contém um único elemento x , ou seja, $|X| = 1$.

Nesse caso, é trivial notar que, pela hipótese inicial, x tem pelo menos um vizinho em Y , digamos y .

Então, $(x, y) \in E$ uma aresta do emparelhamento M que satura todo x (x é único)

Portanto, $P(1)$ é verdade.

Passo de indução: $P(k-1) \rightarrow P(k)$ para k arbitrário.

Supor que para X arbitrário com $|X| < k$, existe M que satura todo x em X . Queremos mostrar que isso implica que para X com $|X| = k > 1$, existe M' que satura todo x em X .

Seja $x \in X$ arbitrário. Então, pela hipótese inicial (q), existe $y \in Y$, tal que $(x, y) \in E$. Seja H o subgrafo induzido pelo conjunto de vértices $(X - x) \cup (Y - y)$, ou seja:

$H = G[(X - x) \cup (Y - y)]$ (H tem esses vértices e todas as arestas de G com extremidades neles)

Note que $|X - x| < |X| = k$.

Caso 1: Neste caso, supomos que existe um emparelhamento M' que satura todo vértice de $X - x$. Assim, fazendo

$$M = M' + (x, y)$$

temos um emparelhamento que satura todo vértice de X . Provamos que $P(k-1) \rightarrow P(k)$.

Caso 2: Neste caso não é possível saturar todos os vértices de $X - x$ pois

$$\exists S' \in X - x \quad |S'| > |N_H(S')|$$

Defina um outro grafo induzido pelo conjunto de vértices $S' \cup N_G(S')$, chamado F_1 :

$$F_1 = G[S' \cup N_G(S')]$$

Note que S' é um subconjunto de $X - x$ e portanto também é um subconjunto de X .

Pela hipótese de indução, a cardinalidade de S' deve ser menor ou igual a cardinalidade da vizinhança de S' em G , ou seja: $|S'| \leq |N_G(S')|$.

Precisamos garantir que S' tem menos elementos que X , ou seja, $|S'| < |X| = k$. Isso é trivial, pois S' é subconjunto de $X - x$ (tem um elemento a menos), ou seja, no máximo $|S'| = k - 1$.

A última pergunta antes de aplicarmos a hipótese de indução é: será que o grafo F_1 satisfaz a condição de Hall? A resposta é SIM. Porque? S' é um subconjunto de X e o grafo G satisfaz a condição de Hall, ou seja, para todo subconjunto S' de X , $|S'| \leq |N_G(S')|$.

Podemos invocar a hipótese de indução e concluir que $\exists M'$ que satura todo x em S' no grafo F_1

Agora, note que:

1. $|S'| \leq |N_G(S')|$
2. $|S'| > |N_H(S')|$

A cardinalidade de S' passa de menor ou igual que a cardinalidade de seus vizinhos em G para maior que a cardinalidade de seus vizinhos em H . O que muda de $N_G(S')$ para $N_H(S')$ é apenas o vértice y que foi deletado de G , o que implica dizer que em G , $|S'| = |N_G(S')|$ (ao reduzir em uma unidade passou a ser estritamente maior). Se fosse menor, não poderíamos inverter a desigualdade com apenas um vértice.

Seja F_2 o grafo induzido pelo conjunto de vértices $(X - S') \cup (Y - N_G(S'))$, ou seja:

$$F_2 = G[(X - S') \cup (Y - N_G(S'))]$$

Repare que esse grafo contém tudo o que ainda resta ser emparelhado, uma vez que encontramos um emparelhamento M' que satura todo x em S' no grafo F_1 .

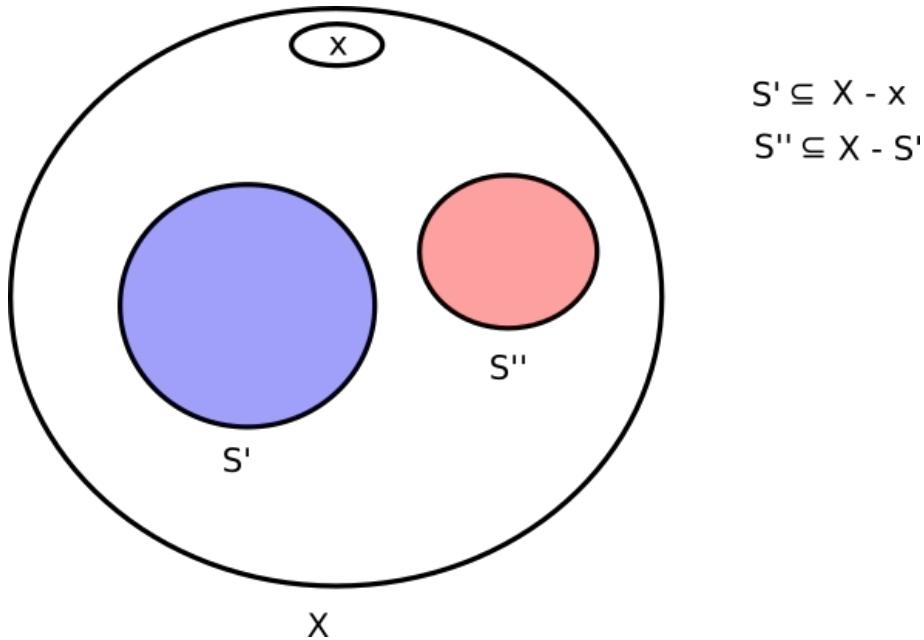
Note que para criar F_2 , reduzimos o número de elementos em S' e o número de elementos de $N_G(S')$ do mesmo valor, uma vez que $|S'| = |N_G(S')|$. Neste ponto, deve ser óbvio que

$|X - S'| < |X| = k$ (pois S' é não vazio) e $|X - S'| = |Y - N_G(S')|$. O que nos resta agora é mostrar que o grafo F_2 satisfaz a condição de Hall. Para isso, seja $S'' \subseteq X - S'$ um subconjunto arbitrário. Desejamos mostrar que a cardinalidade de S'' é menor ou igual a cardinalidade de $N_{F_2}(S'')$ (isso significa que a condição de Hall é válida).

Note que o conjunto $S' \cup S''$ é um subconjunto de X . Pela hipótese inicial, sabemos que:

$$|S' \cup S''| \leq |N_G(S' \cup S'')| \quad (\text{pois } q \text{ é verdade em } q \rightarrow p) (*)$$

Como $S' \subseteq X - x$ e $S'' \subseteq X - S'$, temos que S' e S'' são subconjuntos disjuntos.



Isso implica que

$$|S' \cup S''| = |S'| + |S''|$$

Note ainda que o lado direito de (*) pode ser expresso como:

$$|N_G(S' \cup S'')| = |N_G(S')| + |N_{F_2}(S'')|$$

pois dessa forma estamos contando todos os vizinhos de S' em G , bem como todos os vizinhos de S'' que não são vizinhos de S' . Podemos garantir que não estamos contando duas vezes vizinhos em comum de S' e S'' (que estariam na intersecção) pois por construção subtraímos os vizinhos de S' do grafo F_2 , ou seja, $|N_G(S')|$ e $|N_{F_2}(S'')|$ são conjuntos disjuntos. Isso nos leva a:

$$|S'| + |S''| \leq |N_G(S')| + |N_{F_2}(S'')|$$

Mas como $|S'| = |N_G(S')|$ (verificamos isso anteriormente), temos:

$$|S''| \leq |N_{F_2}(S'')|$$

mostrando que o grafo F_2 satisfaz a condição de Hall. Aplicando a hipótese inicial (q), podemos concluir que existe um emparelhamento M'' que satura todo x em $X - S'$ no grafo F_2 .

Portanto, o emparelhamento $M = M' \cup M''$ satura todo vértice x de X no grafo G . A prova está concluída.

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.

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:
 - a raiz x_0 é M-não-saturada
 - $\forall v \in T$ 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:

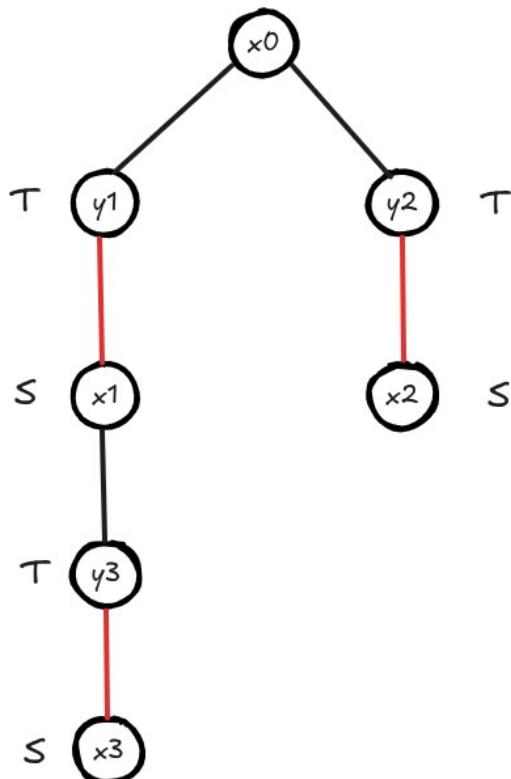
- Árvore do tipo-I: todos os vértices de T são M-saturados (com exceção da raiz x_0)
- Árvore do tipo-II: T possui um vértice folha M-não-saturado

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

Análise das árvores do tipo-I

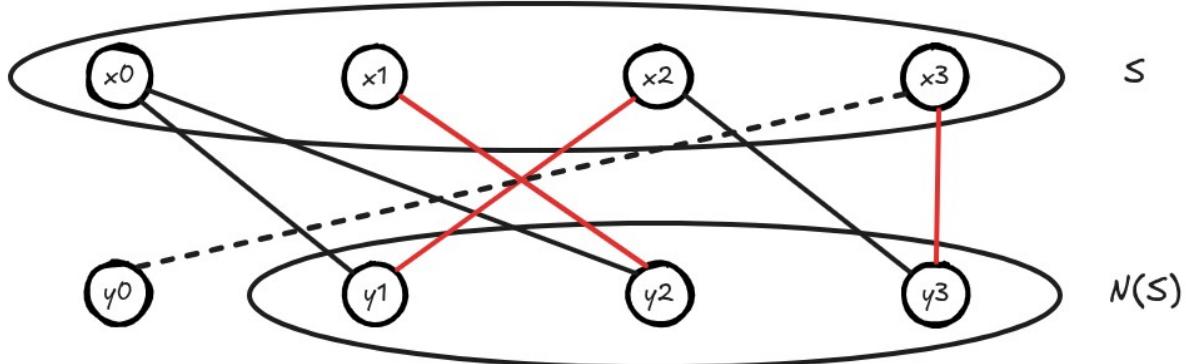
Sejam

- S' : conjunto dos vértices a uma distância par de x_0 (raiz)
- 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 , pois $N(S)$ engloba todos os y que estão na árvore e mais alguns que ainda podem não ter sido “pendurados”.



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)

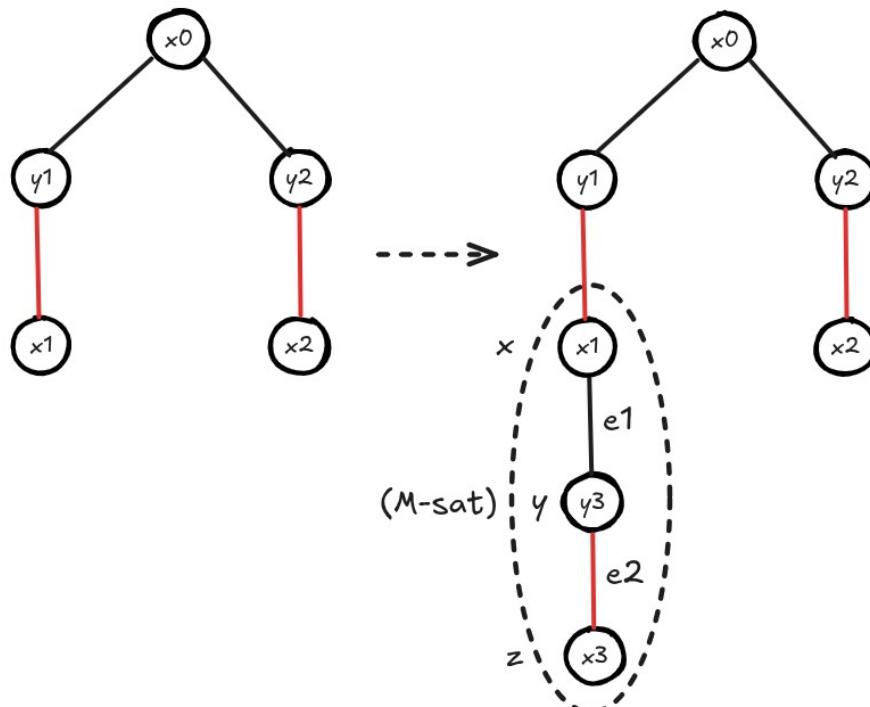
Subcaso a): condição de parada

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): existe caminho M -aumentado, continuar buscando

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

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:

- Tipos-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

Tipos-II Encontramos caminho M-aumentado P: $M' = T(P)$

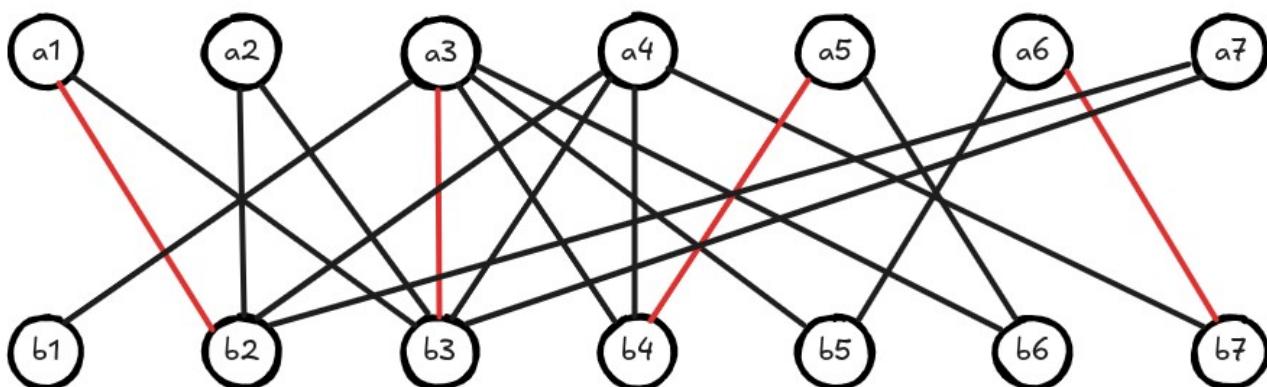
Algoritmo Húngaro

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 *Google* consiste num conjunto de 7 tarefas (a1, a2, a3, a4, a5, a6, a7). A empresa possui na equipe de desenvolvimento apenas 7 profissionais (b1, b2, b3, b4, b5, b6, b7). 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.



$$M^{(0)} = \{a1b2, a3b3, a5b4, a6b7\}$$

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

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

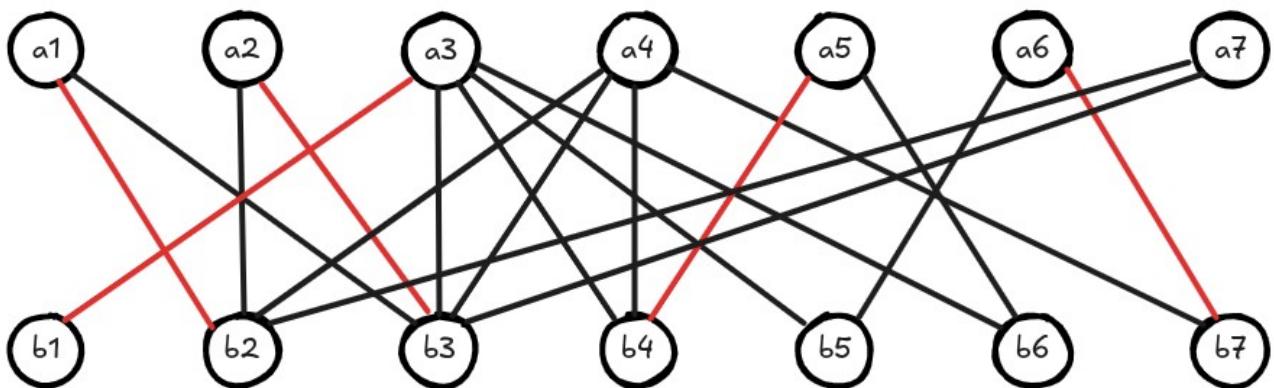
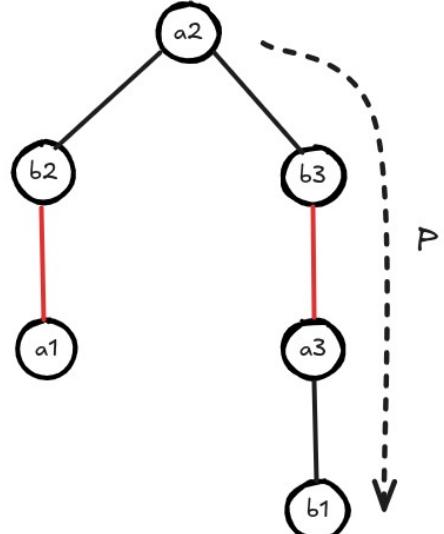
$M^{(1)} = T(P)$ (percorrer P ligando as arestas que não estão em $M^{(0)}$ e desligando as que estão)

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	---

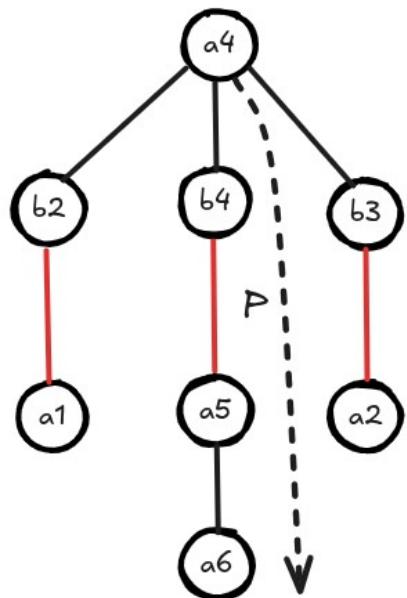
$$P' = b6 \ a5 \ b4 \ a4 \rightarrow P = \text{inv}(P') = a4 \ b4 \ a5 \ b6$$

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

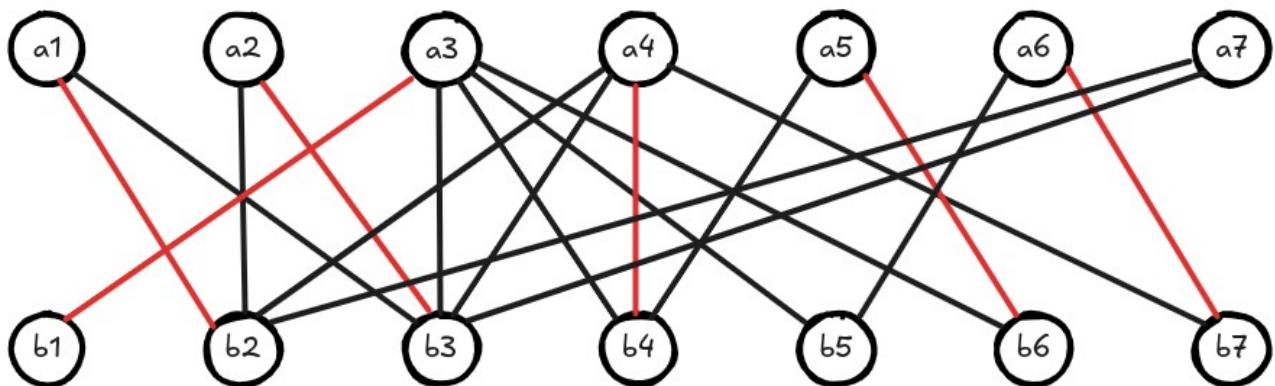
a4 b4 = OK

b4 a5 = x

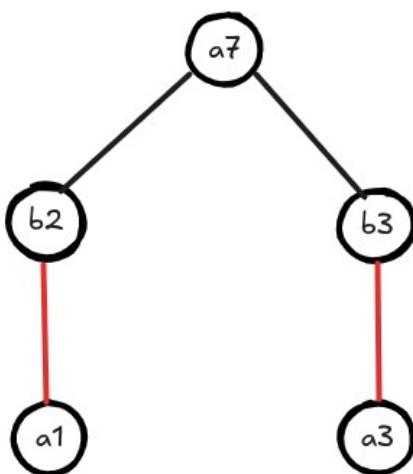
a5 b6 = OK



$$M^{(2)} = \{a_1b_2, a_2b_3, a_3b_1, a_4b_4, a_5b_6, a_6b_7\}$$

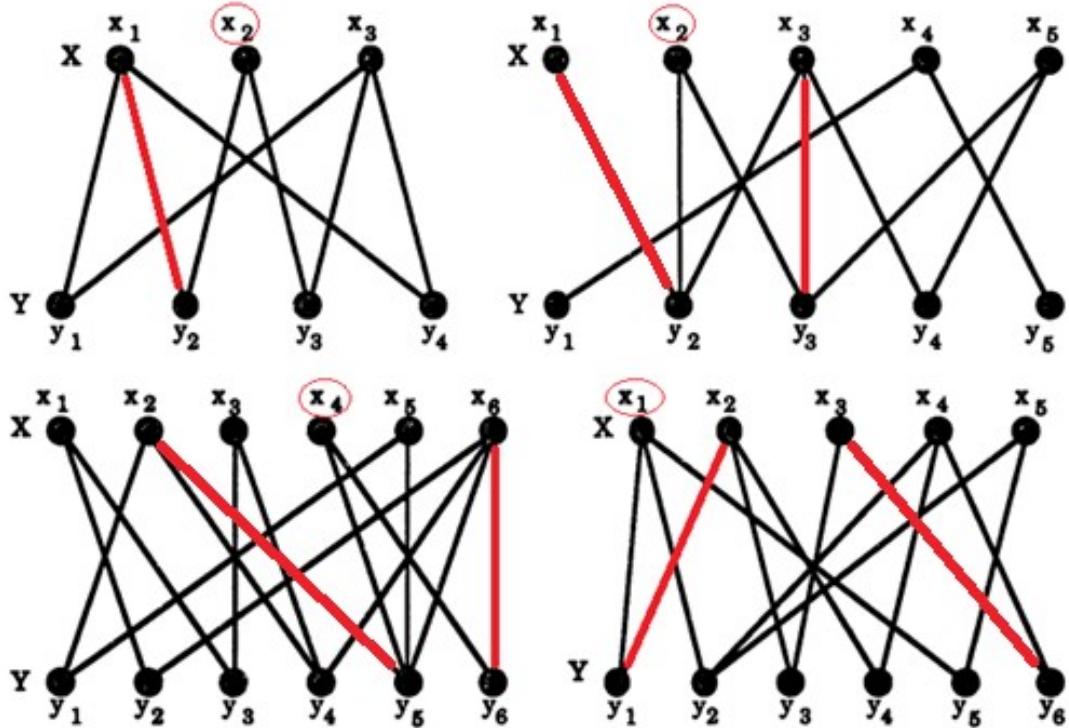


i	S	T	N(S)	$y \in N(s) \wedge y \notin T$	$c_y \in M(z)$
3	[a7]	\emptyset	[b2, b3]	b2 (M-sat)	(b2, a1)
	[a1, a7]	[b2]	[b2, b3]	b3 (M-sat)	(b3, a2)
	[a1, a2, a7]	[b2, b3]	[b2, b3]	---	---



Como $N(S) = T$, temos que $S = \{a_1, a_2, a_3\}$ fere o T. C. pois $N(S) = \{b_2, b_3\}$. Portanto, não há emparelhamento M que sature todo vértice de X . (se houvesse aresta a_7b_7 , OK)

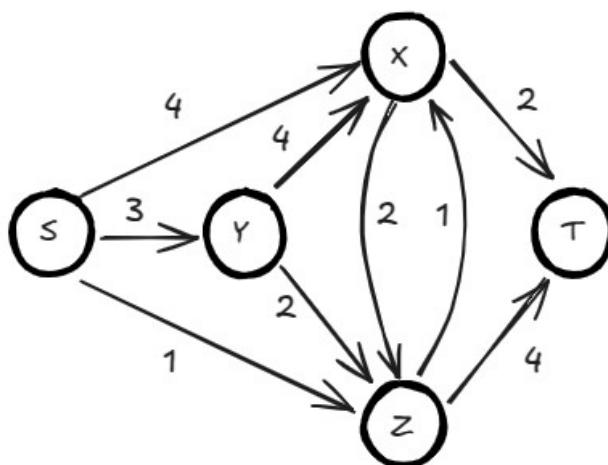
Ex: Utilizando o algoritmo Húngaro, encontre emparelhamentos máximos para os grafos bipartidos a seguir, ou mostre que ele não existe, indicando um subconjunto S de X que viola o Teorema do Casamento.



Fluxo em redes

O estudo de **fluxo em redes** trata da maximização da quantidade de fluxo que pode ser transportado de uma origem para um destino em um grafo direcionado com capacidades associadas às arestas. Esse problema tem aplicações diretas em logística, engenharia, telecomunicações, redes elétricas e alocação de recursos. Um dos resultados mais notáveis dessa área é o **Teorema Min-Cut/Max-Flow**, que estabelece uma conexão profunda entre o fluxo máximo que pode ser enviado da origem ao destino e a capacidade mínima de corte — o menor total de capacidades das arestas que, se removidas, separariam a origem do destino. Esse teorema não apenas fundamenta algoritmos clássicos como o de **Ford-Fulkerson**, mas também oferece uma poderosa dualidade entre otimização e separação em grafos. A teoria de fluxo em redes revela como estruturas aparentemente simples podem esconder problemas computacionais ricos e soluções elegantes, sendo uma das contribuições mais impactantes da teoria dos grafos aplicada.

Motivação: maximizar o fluxo que pode ser produzido pela fonte s e consumido pelo terminal t .



Consideraremos grafos direcionados $G = (V, E, c)$ onde $c: E \rightarrow N^+$ é a capacidade de uma aresta. Capacidade é a quantidade máxima de informação que pode ser transmitida por 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)

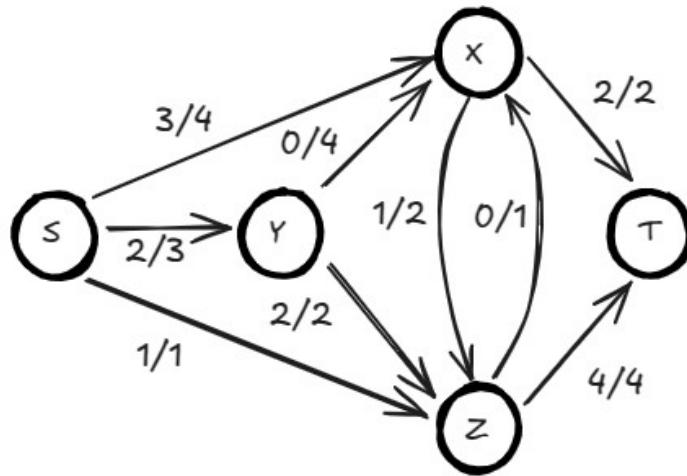
iii) $\forall v \in V - \{s, t\}$ (nó interno)

$$\sum_{e \in I(v)} f(e) = \sum_{e \in O(v)} f(e) \quad (\text{conservação do fluxo})$$

iv) Limite superior: para o valor de fluxo máximo que pode circular em G

$$v(f) = \sum_{e \in O(s)} c(e)$$

Como exemplo, verifique que um fluxo válido para o grafo anterior é representado pela figura a seguir.



Basta verificar as propriedades:

i) OK, pode-se ver facilmente que $\forall e \in E, f(e) \leq c(e)$

ii) $\sum_{e \in O(s)} f(e) = 1 + 2 + 3 = 6 = 2 + 4 = \sum_{e \in I(t)} f(e)$ (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.

Notação:

$$f^{\text{in}}(v) = \sum_{e \in I(v)} f(e) \quad f^{\text{out}}(v) = \sum_{e \in O(v)} f(e)$$

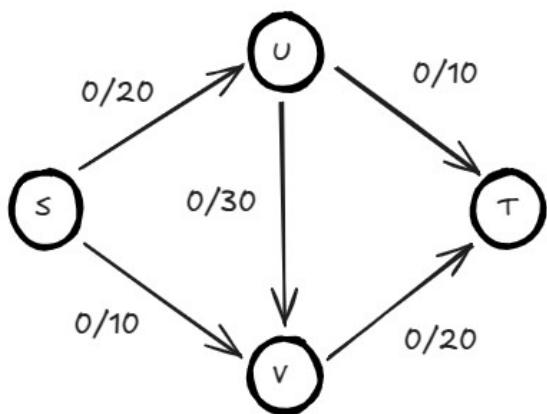
O Problema do Fluxo Máximo

Dado $G = (V, E, c)$ qual o máximo valor de $v(f)$ que pode chegar em t ?

Ideia geral

1. Condição inicial: $f(e) = 0, \forall e \in E$
2. Iteração: encontrar um caminho $s-t$ e transmitir fluxo
3. Condição de parada: Todo caminho $s-t$ encontra-se saturado

Exemplo:



Caso 1. $P = s \cup v \cup t$
 $v(f) = 20$

Caso 2. $P_1 = s \cup t \quad v(f) = 10$
 $P_2 = s \cup v \cup t \quad v(f) = 10 + 10 = 20$
 $P_3 = s \cup v \cup t \quad v(f) = 20 + 10 = 30$

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
 $\forall e \in E$ tal que $f(e) < c(e), \exists$ em E_f uma aresta e' com capacidade residual $c(e') = c(e) - f(e)$ (exatamente o que falta para saturar)
 - b) Backward-Edge (BE): na direção contrária a aresta e
 $\forall e \in E$ tal que $f(e) > 0, \exists$ em E_f uma aresta e'' com capacidade residual $c(e'') = f(e)$ (exatamente o que já passa)

OBS: Note que no início não existem Backward-Edges pois os fluxos iniciais são nulos

RESUMO

1. $\forall e \in E$ com $f(e) = 0$ gera apenas Forward-Edge e'
2. $\forall e \in E$ com $f(e) = c(e)$ gera apenas Backward-Edge e''
3. $\forall e \in E$ com $0 < f(e) < c(e)$ gera 2 arestas: e' (FE) e e'' (BE)

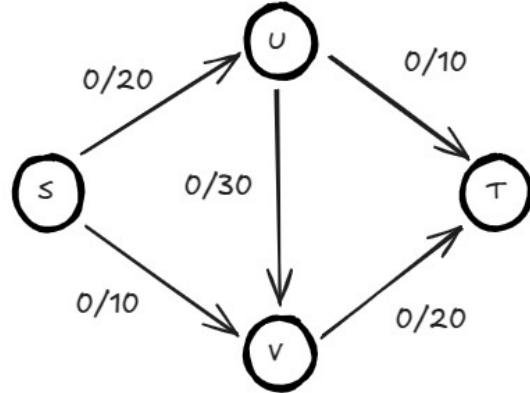


Caminhos aumentados

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 dada por:

```
# melhora fluxo f utilizando o caminho P em G_f
Augment(f, P) {
    b = gargalo(P)      # é o máximo que podemos passar por P
    for each e = (u, v) in P {
        if e está a favor do fluxo s-t em G (F.E.)
            f(e) = f(e) + b
        else (B.E.)
            f(e) = f(e) - b
    }
    return f
}
```

Exemplo:



Passo 1: Grafo residual é o próprio G

$$f = 0$$

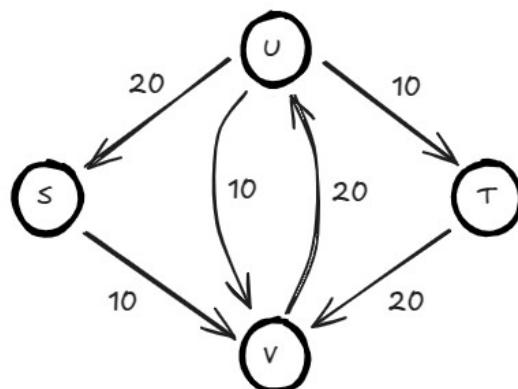
$$P = suvt$$

$$b = 20$$

$$f(su) = 0 + 20 = 20$$

$$f(uv) = 0 + 20 = 20$$

$$f(vt) = 0 + 20 = 20$$



Passo 2:

$$f = 20$$

$$P = svut$$

$$b = 10$$

$$f(sv) = 0 + 10 = 10$$

Contrária a aresta (u, v)

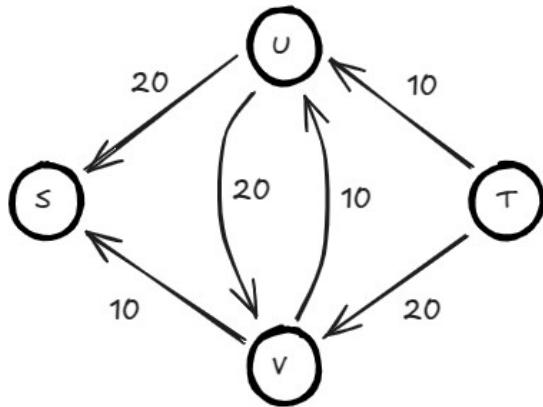
$$f(vu) = f(uv) - 10 = 20 - 10 = 10$$

Passo 3:

$$f = 30$$

$\nexists P_{st}$ em $G_f \rightarrow$ PARE

\Rightarrow O fluxo máximo vale 30



Teorema: O resultado da operação $f' = \text{Augment}(f, P)$ é um fluxo válido.

i) f' não excede a capacidade (no caso de F.E.)

$$f'(e) = f(e) + b \leq f(e) + (c(e) - f(e))$$

pois para qualquer aresta F.E. do caminho $c(e) - f(e)$ será maior ou igual que b

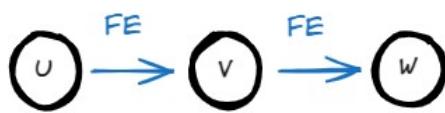
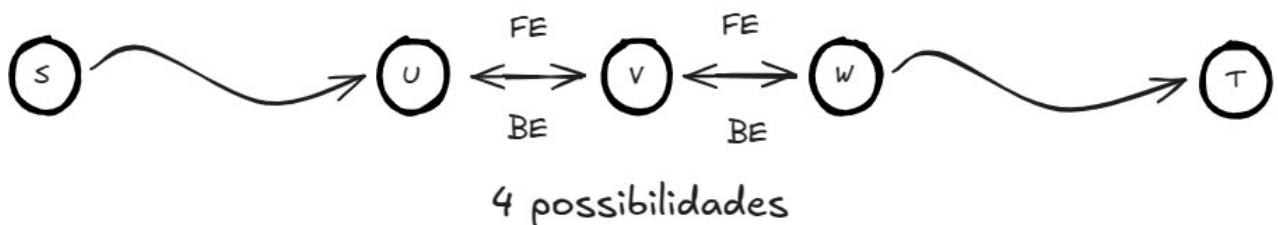
ii) f' nunca é inferior a zero (no caso de B.E.)

$$f(e) \geq f'(e) = f(e) - b \geq f(e) - f(e) = 0$$

pois para qualquer aresta B.E. do caminho $f(e)$ será maior ou igual que b

iii) conservação (tudo que entra em v sai de v)

$$f^{\text{in}}(v) = f^{\text{out}}(v) \quad \text{para todo nó interno } v$$



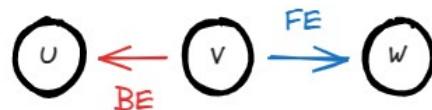
+ b na entrada
+ b na saída



+ b na entrada
- b na entrada



- b na entrada
- b na saída



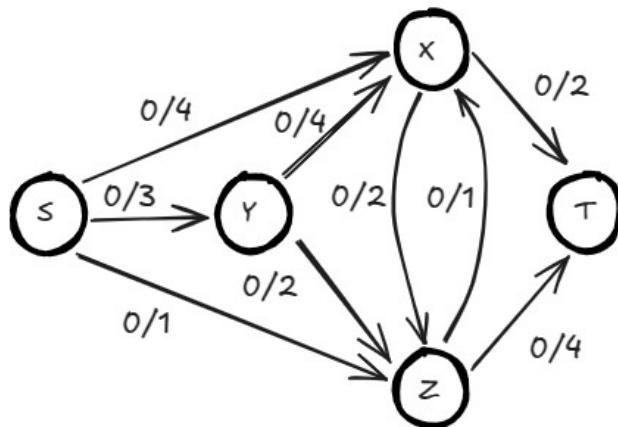
- b na saída
+ b na saída

mostrando que em qualquer um dos casos, a conservação do fluxo é preservada!

Portanto, a operação Augment(f, P) preserva fluxos e podemos usá-la para melhorar um dado fluxo. Veremos a seguir uma sequência lógica de passos para obter o fluxo máximo em um grafo: o algoritmo de Ford-Fulkerson, um algoritmo guloso pois tenta passar o máximo de fluxo possível em cada caminho aumentado.

```
Max_Flow(G, s, t, c) {
    f = 0
    for each e in E
        f(e) = 0
    while ∃Pst in Gf {
        Seja Pst um caminho Pst no grafo residual
        f' = Augment(f, Pst)
        Atualize o grafo residual Gf
    }
}
```

Exemplo: Utilizando ao algoritmo de Ford-Fulkerson, encontre o fluxo máximo



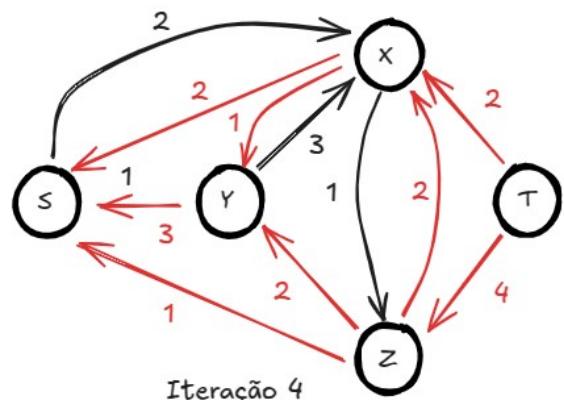
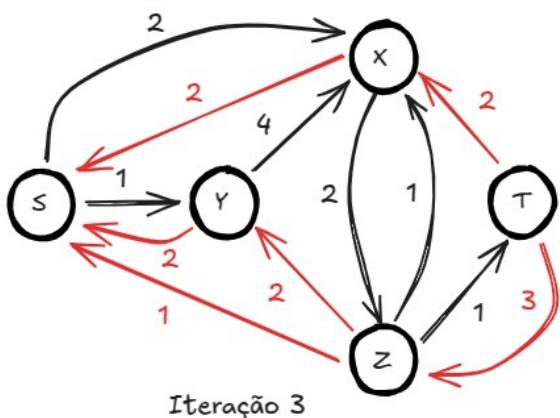
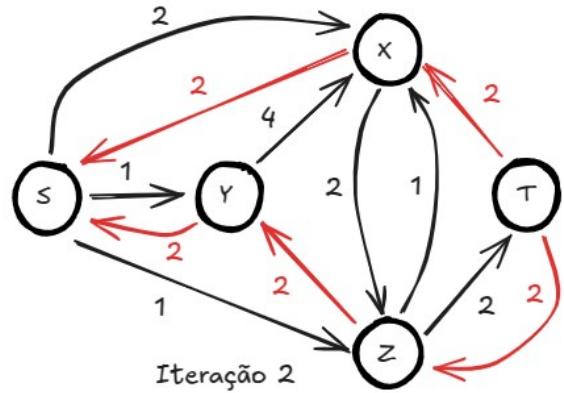
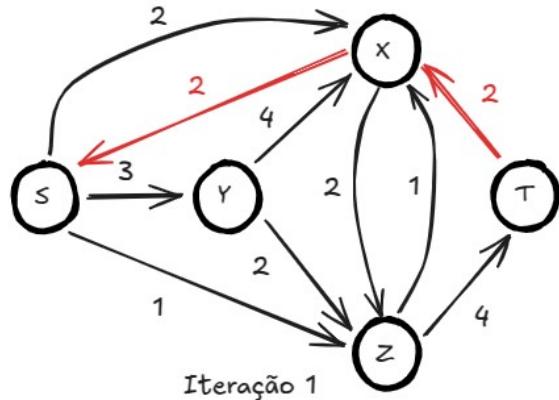
Trace do algoritmo

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	$f(sy) = 2 + 1 = 3$ $f(yx) = 0 + 1 = 1$ $f(xz) = 0 + 1 = 1$ $f(zt) = 3 + 1 = 4$	$5 + 1 = 6$

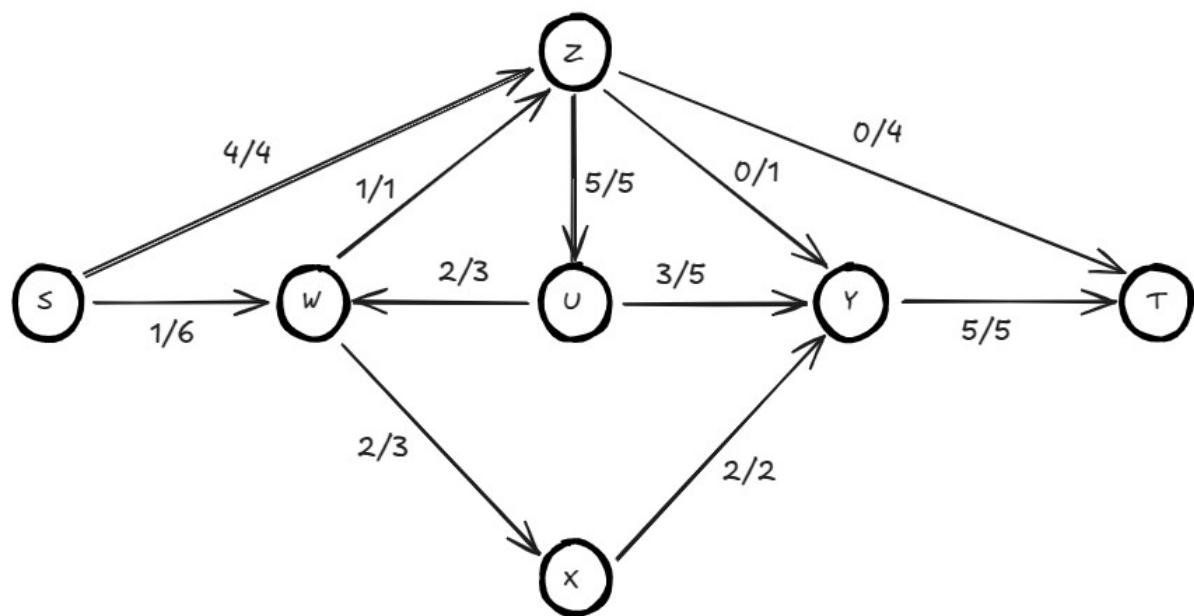
Portanto, o fluxo máximo vale 6.

A seguir encontram-se os grafos residuais após cada um dos passos do algoritmo.

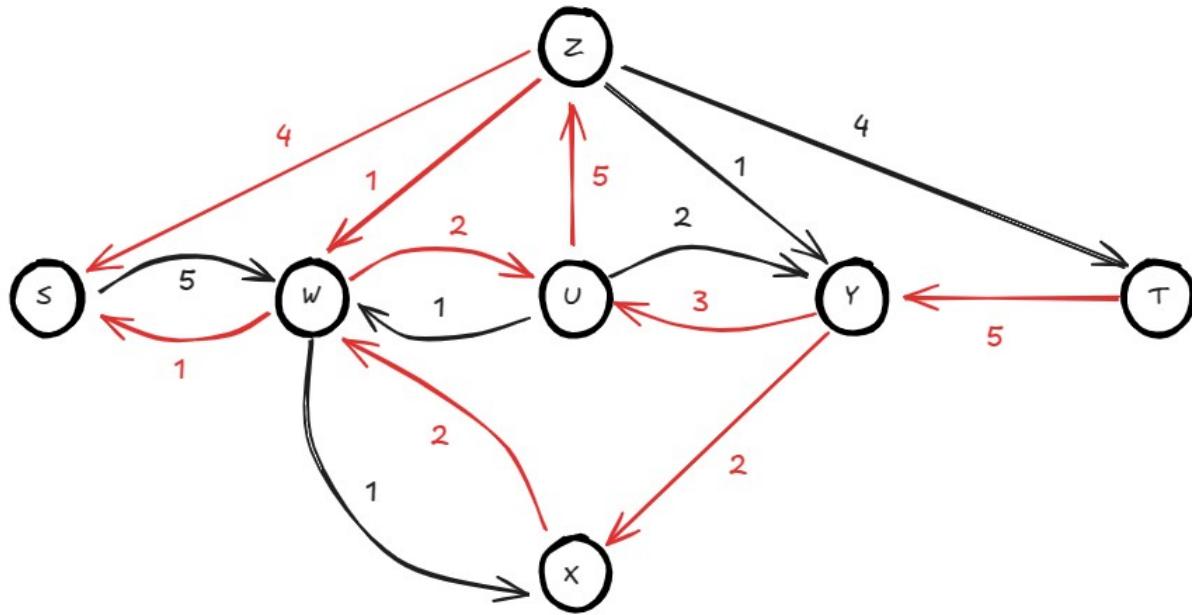
Note que a cada iteração, após passarmos um fluxo pelo caminho st , o grafo residual é modificado para refletir as novas capacidades restantes.



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:

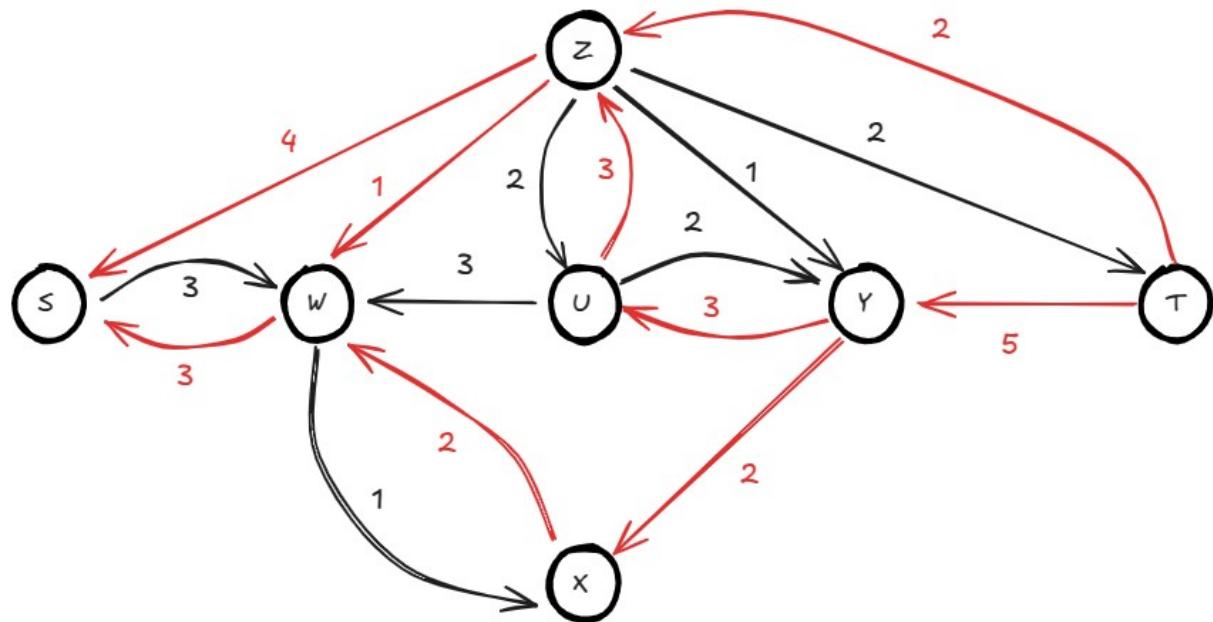
$$f(sw) = 1 + 2 = 3, \text{ pois a aresta } (s,w) \text{ está alinhada ao fluxo em } G$$

$$f(wu) = 2 - 2 = 0, \text{ pois a aresta } (w,u) \text{ está ao contrário do fluxo em } G$$

$$f(uz) = 5 - 2 = 3, \text{ pois a aresta } (u,z) \text{ está ao contrário do fluxo em } G$$

$$f(zt) = 0 + 2 = 2, \text{ pois a aresta } (z,t) \text{ está alinhada ao fluxo em } G$$

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.

Fluxos e cortes

Aplicações: segmentação de imagens, classificação supervisionada, emparelhamentos

Motivação: cortes especificam limites superiores para $v(f)$ (valor do fluxo máximo). A relação entre fluxo e cortes nos permite resolver um problema NP-Hard (corte mínimo em grafos) em tempo polinomial.

Ideia: Ao partitionar G em 2 componentes A e B , a capacidade das arestas cortadas impõe um limite superior para o fluxo que pode sair de A e chegar em B .

Def: Um corte $s-t$ em G é qualquer partição de V em A e B tal que $s \in A$ e $t \in B$

Def: A capacidade de um corte é definida como:

$$c(A, B) = \sum_{e \in O(A)} c(e) \quad (\text{soma das capacidades das arestas que saem de } A \text{ e chegam em } B)$$

Min-cut/Max-flow: resultado que mostra a equivalência entre 2 problemas de otimização duais.

=> Encontrar o fluxo máximo corresponde a encontrar o corte de mínima capacidade (2 problemas que podem ser resolvidos com um único algoritmo)

Para provar o teorema Min-cut/Max-flow, iremos primeiramente demonstrar uma série de resultados preliminares importantes que serão utilizados depois.

Teorema: Seja f um fluxo $s-t$ e (A, B) um corte $s-t$. Então:

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \quad (\text{ou seja, o valor do fluxo depende diretamente do corte: tudo que sai de } A \text{ menos tudo que entra em } A)$$

$$v(f) = \sum_{e \in O(s)} f(e) = f^{\text{out}}(s)$$

Sabe-se que $f^{\text{in}}(s) = 0$ então podemos escrever $v(f) = f^{\text{out}}(s) - f^{\text{in}}(s)$

Mas também sabemos que $\forall v \in V - \{s, t\}$ (nó intermediário), pela conservação do fluxo vale:

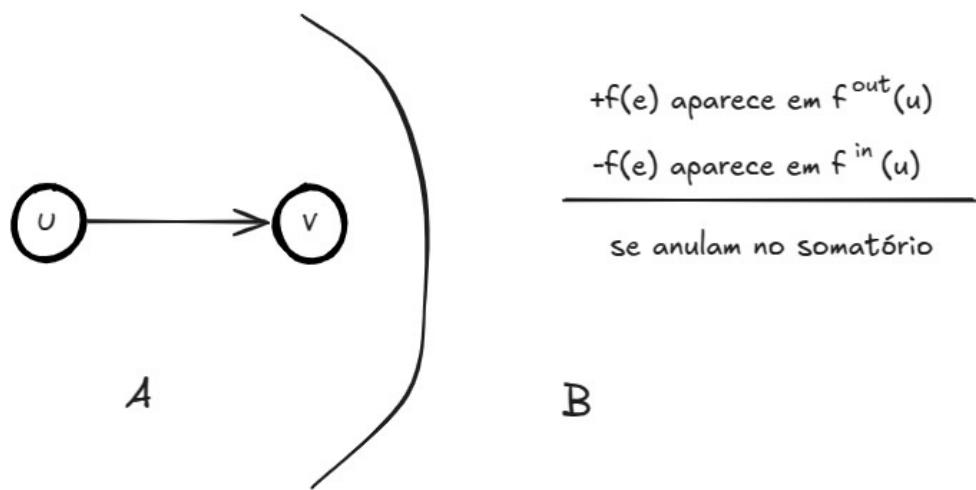
$$f^{\text{in}}(v) = f^{\text{out}}(v) \Leftrightarrow f^{\text{out}}(v) - f^{\text{in}}(v) = 0$$

Desse modo, podemos expressar $v(f)$ como:

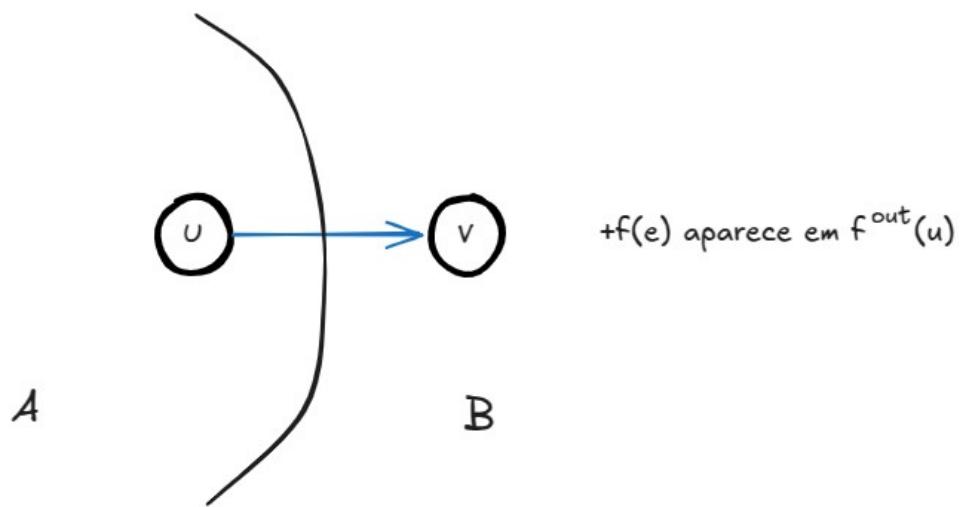
$$v(f) = \sum_{v \in A} (f^{\text{out}}(v) - f^{\text{in}}(v)) \quad (\text{todos os termos desse somatório são nulos, exceto para } s)$$

Porém, existem apenas 4 tipos de arestas em G :

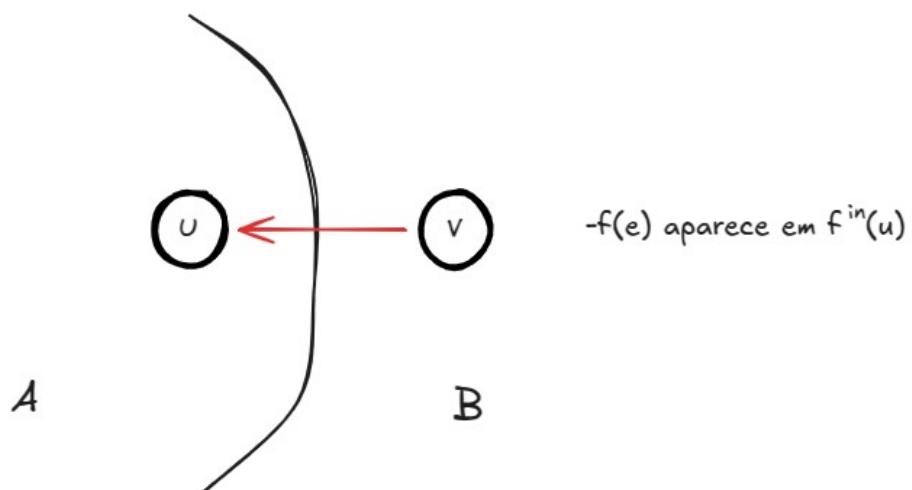
i) $e = \langle u, v \rangle$ com $u, v \in A$



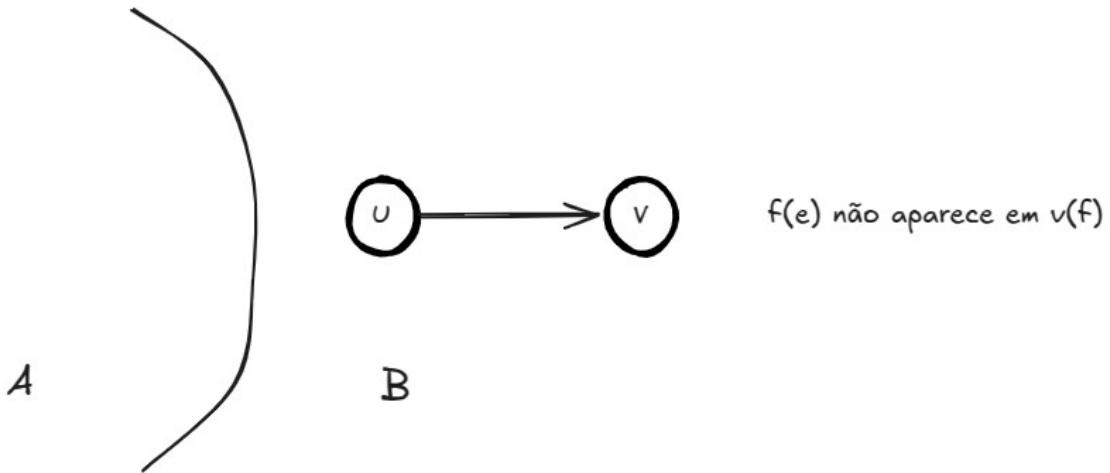
ii) $e = \langle u, v \rangle$ com $u \in A$ e $v \in B$



iii) $e = \langle v, u \rangle$ com $u \in A$ e $v \in B$



iv) $e = \langle u, v \rangle$ com $u, v \in B$



Portanto, $v(f)$ pode ser expresso por:

$$v(f) = \sum_{e \in O(A)} f(e) - \sum_{e \in I(A)} f(e) = f^{\text{out}}(A) - f^{\text{in}}(A)$$

OBS: Note que $f^{\text{out}}(A) - f^{\text{in}}(A) = f^{\text{in}}(B) - f^{\text{out}}(B)$

Teorema: Seja um fluxo s-t qualquer e (A, B) um corte s-t. Então, $v(f) \leq c(A, B)$

$$v(f) = f^{\text{out}}(A) - f^{\text{in}}(A) \leq f^{\text{out}}(A) = \sum_{e \in O(A)} f(e) \leq \sum_{e \in O(A)} c(e) = c(A, B)$$

OBS:

a) Se encontrarmos um fluxo s-t máximo f^* , não existe corte $c(A, B)$ com capacidade menor que $v(f^*)$.

b) Se encontrarmos um corte s-t (A, B) com capacidade mínima c^* , não existe fluxo s-t com valor maior que c^*

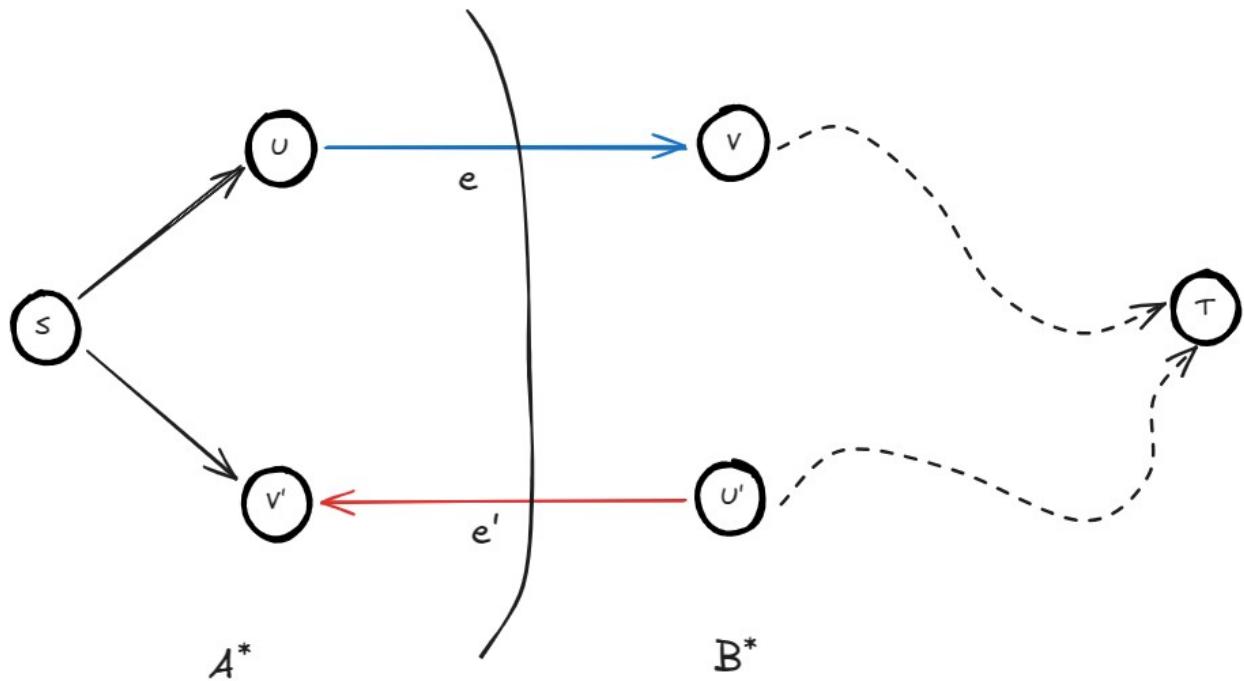
Teorema (Min-cut/Max-flow): Seja f^* o fluxo s-t tal que $\nexists P_{st}$ em G_f (fluxo gerado pelo algoritmo Ford-Fulkerson). Então, existe um corte (A^*, B^*) para qual o valor $v(f^*) = c(A^*, B^*)$. Consequentemente, f^* tem máximo valor de fluxo e o corte (A^*, B^*) tem a mínima capacidade dentre todos os possíveis. (encontrar fluxo máximo nos dá corte mínimo, o que era NP-hard).

Prova:

Seja $G_f = (V_f, E_f)$ o grafo residual de G no momento em que não existe mais caminho P_{st} . Podemos então dividir o conjunto V em 2 partições:

$$A^* = \{v \in V_f / \exists P_{sv}\} \quad (\text{atingíveis a partir de } s)$$

$$B^* = V_f - A^* \quad (\text{não atingíveis a partir de } s)$$



Note que:

a) $t \in B^*$ pois $\nexists P_{st}$ em G_f

b) Vamos analisar a aresta $e = \langle u, v \rangle$. Essa aresta certamente está saturada, ou seja, $f(e) = c(e)$, pois caso contrário haveria caminho P_{sv} uma vez que a capacidade residual da aresta F.E. correspondente em G_f seria $c(e) - f(e)$. Assim, podemos generalizar essa observação e escrever o seguinte fato:

$$\forall e \in O(A^*) \text{ tem } f(e) = c(e) \text{ (todas as arestas que saem de } A \text{ são saturadas)}$$

c) Análise da aresta $e' = \langle u', v' \rangle$. Essa aresta certamente não carrega fluxo algum ($f(e)$ nulo), pois caso contrário seria gerado uma aresta $e'' = \langle v', u' \rangle$ em G_f (B.E.) com capacidade residual $f(e')$, o que resultaria num caminho $P_{su'}$. Assim, podemos generalizar essa observação para:

$$\forall e \in I(A^*) \text{ tem } f(e) = 0 \text{ (todas as arestas que entram em } A \text{ tem fluxo nulo)}$$

Em resumo, se (A^*, B^*) é um corte, então:

$$\forall e \in O(A^*) , f(e) = c(e)$$

$$\forall e \in I(A^*) , f(e) = 0$$

Calculando o valor do fluxo temos:

$$v(f) = f^{\text{out}}(A^*) - f^{\text{in}}(A^*) = \sum_{e \in O(A^*)} f(e) - \sum_{e \in I(A^*)} f(e) = \sum_{e \in O(A^*)} c(e) = c(A^*, B^*) \quad (\text{min. capac. corte})$$

OBS: A saída do Ford-Fulkerson, $v(f)$, é de fato o valor de $c(A^*, B^*)$. Os conjuntos A^* e B^* são obtidos por uma simples busca em largura a partir de s em G_f .

Introdução as redes complexas

Redes complexas são grafos que representam sistemas reais com topologias não triviais, frequentemente caracterizados por estruturas heterogêneas, agrupamentos locais e padrões de conexão que fogem do comportamento aleatório tradicional. Esses grafos são utilizados para modelar uma ampla gama de sistemas interconectados, como redes sociais, redes biológicas, redes de transporte, sistemas de energia e a própria internet. Ao contrário dos modelos clássicos, as redes complexas frequentemente exibem propriedades como o **pequeno mundo** (small-world effect), onde a maioria dos nós pode ser alcançada a partir de qualquer outro em poucos passos, e **distribuições de grau em lei de potência**, típicas das chamadas redes **livres de escala** (scale-free). Modelos como o de **Erdős-Rényi** (aleatório), **Watts-Strogatz** (pequeno mundo) e **Barabási-Albert** (crescimento com preferência) permitem simular e estudar essas estruturas com base em diferentes mecanismos de formação. O estudo de redes complexas combina teoria dos grafos, estatística e física computacional, oferecendo ferramentas poderosas para compreender a dinâmica, a resiliência e a evolução de sistemas complexos no mundo real.

- Grafos dinâmicos
- Teoria dos grafos + Probabilidade + Estatística: grafos como variáveis aleatórias

Motivação: Estudar propriedades e dinâmica de sistemas complexos encontrados na natureza (moléculas, rede de proteínas, internet, redes sociais)

Existem basicamente 3 modelos principais:

- i) Modelo ER (Erdős-Renyi)
- ii) Modelo de pequeno mundo (Watts-Strogatz)
- iii) Modelo livre de escala (Barabási-Alberts)

O que caracteriza um modelo? Resumidamente 3 aspectos:

- i) Distribuição dos graus
- ii) APL (average path length) – comprimento médio dos caminhos (extensão do grafo)
- iii) Coeficiente de aglomeração (densidade de ligação entre os vizinhos: número de triângulos)

Modelo ER

Primeiro modelo proposto: mais antigo

G é um grafo $ER(n, p)$

n : número de vértices

p : probabilidade de conexão entre 2 vértices

Algoritmo $ER(n, p)$

1. Inicie com o grafo nulo de n vértices
2. Escolha aleatoriamente um par de vértices u e v
3. Se a aresta (u,v) não existe
 - Gere um número aleatório $x \in [0,1]$
 - Se $x \leq p$
 $G = G + (u,v)$
4. Repita os passos 2 e 3 até que $|E|=m$ ou o grau médio seja igual a k

Distribuição dos graus

Em muitos casos é necessário computar qual a probabilidade de um vértice ter k vizinhos. Para isso precisamos saber qual é a distribuição dos graus em um grafo do tipo ER. Pode-se mostrar que em grafos ER essa distribuição é uma Binomial com parâmetros $n-1$ e p .

Binomial(m, p)

$$p(x=k) = \binom{m}{k} p^k (1-p)^{m-k} \quad \text{onde} \quad \binom{m}{k} = \frac{m!}{(m-k)!k!}$$

(probabilidade de ter k sucessos em m tentativas sabendo que probabilidade de 1 sucesso é p)

A ideia é que para um vértice ter grau k é preciso ter k sucessos em um total de $n-1$ vértices.

$$p(d(v)=k) = \binom{n-1}{k} p^k (1-p)^{(n-1)-k} \quad \text{Binomial}(n-1, p)$$

Sabendo que a distribuição dos graus é uma binomial, é possível calcular a média dessa distribuição, que nada mais é que o grau médio ou valor esperado dos graus: $E[k]$ ou $\langle k \rangle$

$$E[k] = \sum_k k p(k)$$

A variância, que mede a dispersão da distribuição, é dada por:

$$\sigma^2 = E[(k - E[k])^2]$$

Problema: determine qual é o valor esperado e a variância da distribuição dos graus do modelo ER

Para facilitar os cálculos, iremos utilizar um conhecido resultado da teoria das probabilidades que diz que uma V.A. Binomial(m, p) é a soma de m V.A.'s de Bernoulli(p), ou seja:

$$\text{Se } Y = \sum_{i=1}^m X_i \quad \text{com} \quad X_i \sim \text{Bernoulli}(p), \text{ então} \quad Y \sim \text{Binomial}(m, p)$$

Nesse caso temos que para $X_i \sim \text{Bernoulli}(p)$:

$$\begin{aligned} p(X_i=0) &= 1-p && (\text{fracasso}) \\ p(X_i=1) &= p && (\text{sucesso}) \end{aligned}$$

Assim, o valor esperado é:

$$E[X_i] = 0 \cdot p(X_i=0) + 1 \cdot p(X_i=1) = p$$

A variância é dada por:

$$\sigma^2 = E[(X_i - E[X_i])^2] = E[X_i^2 - 2X_i E[X_i] + E^2[X_i]] = E[X_i^2] - 2E^2[X_i] + E^2[X_i] = E[X_i^2] - E^2[X_i]$$

Sabemos que $E^2[X_i] = p^2$ e

$$E[X_i^2] = \sum_k k^2 p(X_i=k) = 0^2 p(X_i=0) + 1^2 p(X_i=1) = p$$

de modo que

$$\sigma^2 = p - p^2 = p(1-p)$$

De posse do valor esperado e variância da Bernoulli, voltamos para a Binomial. Usando a linearidade dos operadores $E[]$ e $\text{Var}[]$, podemos escrever:

$$E[Y] = \sum_{i=1}^m E[X_i] = \sum_{i=1}^m p = mp$$

$$\text{Var}[Y] = \sum_{i=1}^m \text{Var}[X_i] = \sum_{i=1}^m p(1-p) = mp(1-p)$$

Mas, como no modelo ER temos uma Binomial($n-1$, p), temos:

$$E[k] = p(n-1)$$

$$\text{Var}[k] = p(1-p)(n-1)$$

Essas equações nos permitem calcular o grau médio de qualquer grafo do modelo ER(n,p) sem a necessidade de gerar amostras e simular o modelo.

Average Path Length (APL)

Usando um modelo aproximado baseado em árvore pode-se mostrar que o comprimento médio dos caminhos no modelo ER é aproximado por:

$$d \approx \frac{\log n}{\log \langle k \rangle} \quad (\text{n é o número de vértices e } \langle k \rangle \text{ é o grau médio})$$

Uma análise gráfica dessa variável mostra que o APL satura quando o número de vértices cresce. Portanto, de modo geral podemos afirmar que o APL em grafos ER é baixo.

Apesar disso, o modelo ER não é adequado para modelar redes sociais reais. O motivo está na baixa densidade de triângulos, ou seja, em um baixo coeficiente de aglomeração.

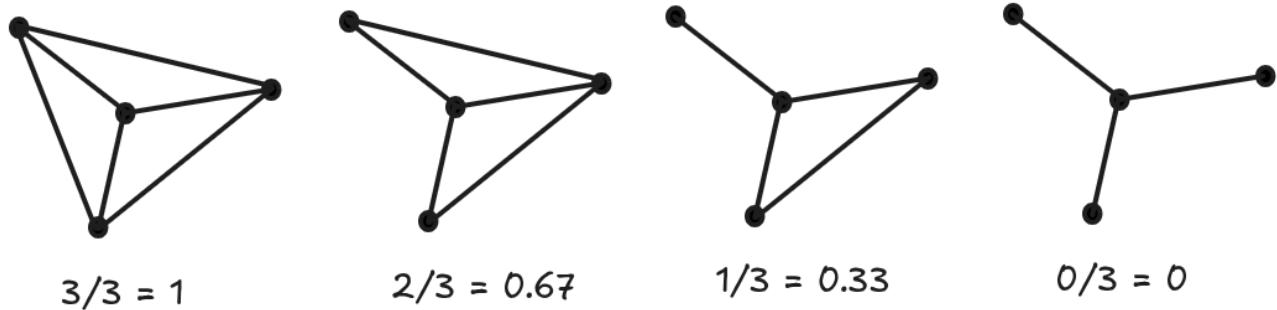
Coeficiente de aglomeração (Clustering coefficient)

Mede o quanto conectados os vizinhos de 1 nó estão

$$cc(u) = \frac{m_u}{\binom{n_u}{2}} \quad \text{onde}$$

m_u é o número real de arestas entre os vizinhos de u

$\binom{n_u}{2}$ é o número de arestas máximas entre os vizinhos de u



O coeficiente de aglomeração tem relação direta com o número de triângulos em G . Quanto mais triângulos, maior.

Para grafos ER(n, p), pode-se mostrar que: $cc(u) = p$

Problema:

p é a probabilidade de uma aresta existir, ou seja, é a probabilidade de 2 indivíduos quaisquer no sistema se conhecerem

$cc(u)$ é a probabilidade de 2 indivíduos de uma vizinhança se conhecerem (é a probabilidade dos amigos de alguém se conhecerem)

Nesse modelo, como p é igual a $cc(u)$, temos que a probabilidade de 2 vizinhos se conhecerem é igual a probabilidade de 2 estranhos se conhecerem (não é algo razoável em termos de refletir o comportamento de sistemas sociais por exemplo). Portanto, o modelo ER não serve para simular o comportamento de redes sociais no mundo real.

Redes de Pequeno Mundo (modelo de Watts-Strogatz)

Modelo otimizado para propagação da informação. Embora distribuição dos graus se pareça com ER, possui uma importante propriedade exclusiva: a propriedade de pequeno mundo (modelo ideal para simular redes sociais – teoria dos 6 graus de separação).

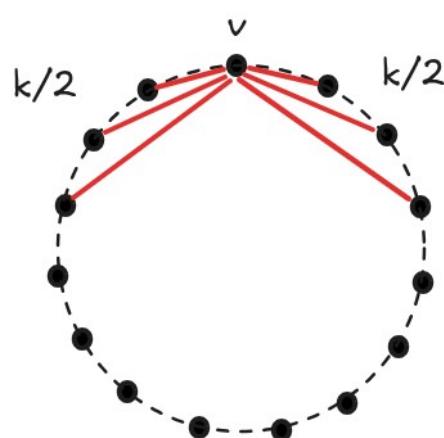
Algoritmo WS(n, k, p)

n : número de vértices

k : número de vizinhos iniciais

p : probabilidade de religação (“rewiring”)

1. Dispor n vértices numa fila circular
2. Ligar cada vértice a seus $k/2$ vizinhos mais próximos a esquerda e a direita



3. Escolher um vértice u aleatoriamente
4. Escolher um vértice v diferente de u aleatoriamente.
5. Se $(u, v) \in E$
 - Escolha um terceiro vértice w diferente de u e v
 - Com probabilidade p desligue a aresta (u, v) e ligue a aresta (u, w) (“rewiring”)
 - (sorteie um número aleatório $x \in [0, 1]$ e se $x < p$ religue)
6. Repetir os passos 3, 4 e 5 até um certo N (max. Iterações) ou até atingir um número desejado de religações.

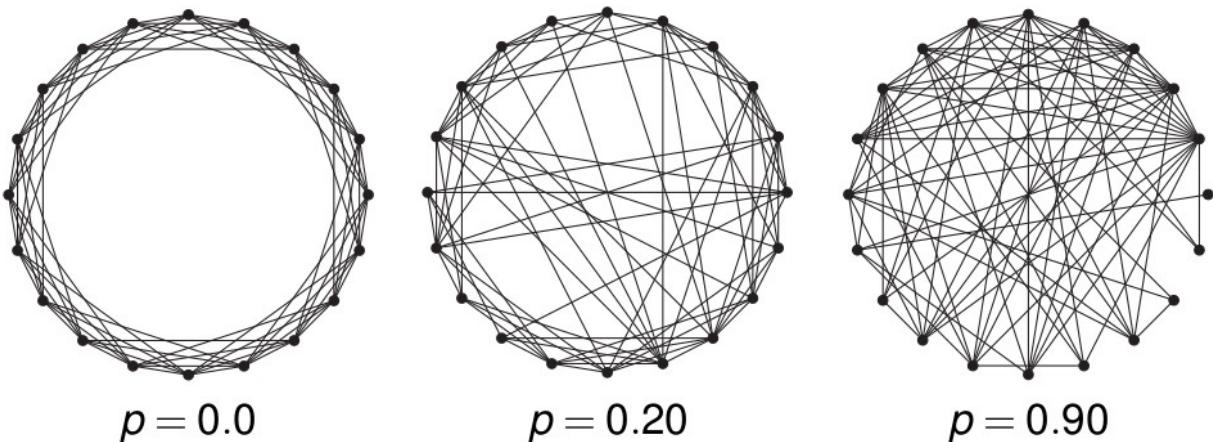
Obs:

- se $p = 0 \rightarrow$ grafo k -regular (grafo inicial não se altera)
- se $p = 1 \rightarrow ER(n, p)$ (troca tudo e vira modelo totalmente aleatório)
- $W(n, k, 1) = ER(n, p)$

Propriedade small-world

Os grafos gerados pelo modelo $WS(n, k, p)$ possuem a interessante propriedade de pequeno mundo, que emerge da combinação de duas propriedades fundamentais:

- a) APL baixo
- b) Coeficiente de aglomeração alto (elevado número de triângulos)



Pode-se mostrar que para grafos $WS(n, k, 0)$ (iniciais) o coeficiente de aglomeração é

$$cc(G) \frac{3}{4} \frac{(k-2)}{(k-1)}$$

o que para K suficientemente grande implica em aproximadamente 75% de todos os possíveis triângulos (é um valor elevado)

Além disso, o comprimento médio dos caminhos em grafos $WS(n, k, 0)$ pode ser aproximado analiticamente, sendo dado por:

$$\bar{d}(u) \approx \frac{(n-1)(n+k-1)}{2kn}$$

o que novamente para n e k suficientemente grande, tende a valores muito baixos.

A dedução desses 2 resultados anteriores pode ser encontrada no livro “Graph Theory and Complex Networks”, capítulo 7, páginas 169 a 171.

Presença de weak links em grafos WS(n, k, p)

Resultado do processo de “rewiring”

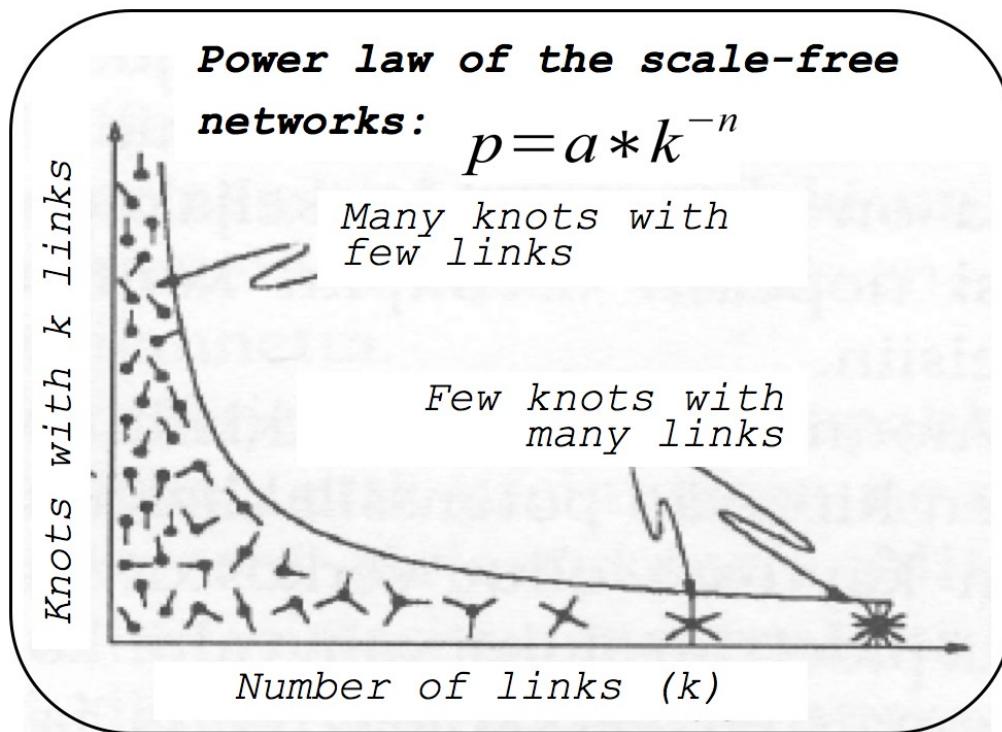
Weak links: arestas de longo alcance (ligam vértices de comunidades distintas)

Papel fundamental na difusão da informação em redes sociais (torna esse modelo muito mais eficiente para propagação de informação que o modelo ER)

Scale-Free Networks (modelo Barabasi-Alberts)

Vimos que o modelo WS é particularmente útil na análise e simulação de redes sociais. Porém, ao se aplicar esse modelo para estudar outros tipos de redes do mundo real, o que se observa é um comportamento que não corresponde a realidade. Um exemplo disso é no estudo da internet.

Várias propriedades como a distribuição dos graus e o padrão de crescimento da internet são muito diferente do que se observa em redes sociais. No modelo BA, a distribuição dos graus segue uma lei de potência (power law), ou seja, há muitos vértices de grau baixo e poucos com grau elevado (são os chamados hubs)



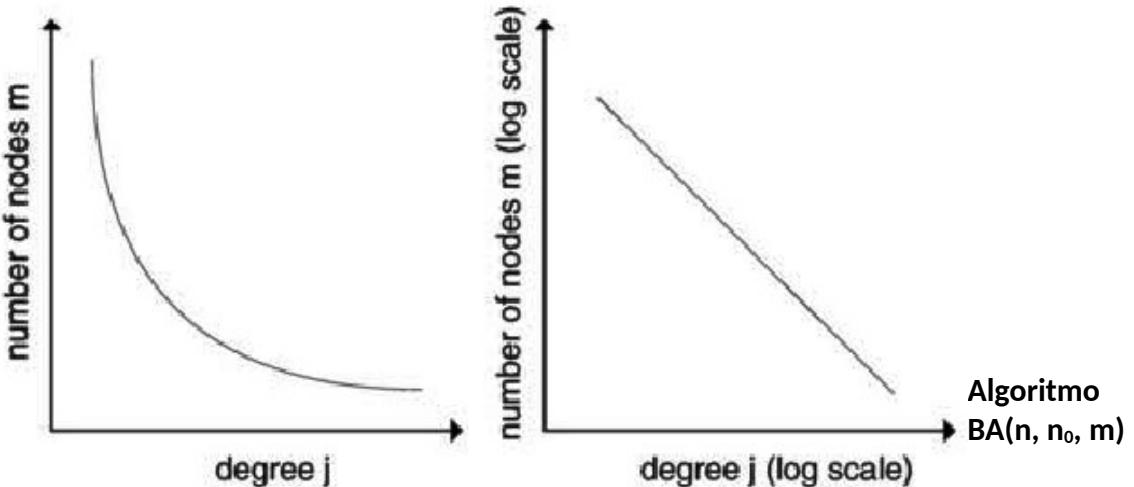
A distribuição dos graus é da forma:

$$p(k) \propto \frac{1}{k^\alpha} \quad \text{com} \quad 2 \leq \alpha \leq 3$$

Aplicando o logaritmo de ambos os lados, temos:

$$\log p(k) = -\alpha \log k$$

o que mostra que um gráfico log-log de uma lei de potência é uma linha reta.



Inicialmente, temos $G_0 = (V_0, E_0)$ com $n_0 = |V_0|$ e E_0 desprezível, onde G_0 é uma amostra do modelo $ER(n_0, p)$

1. Adicione um novo vértice v_s ao conjunto de vértices atual

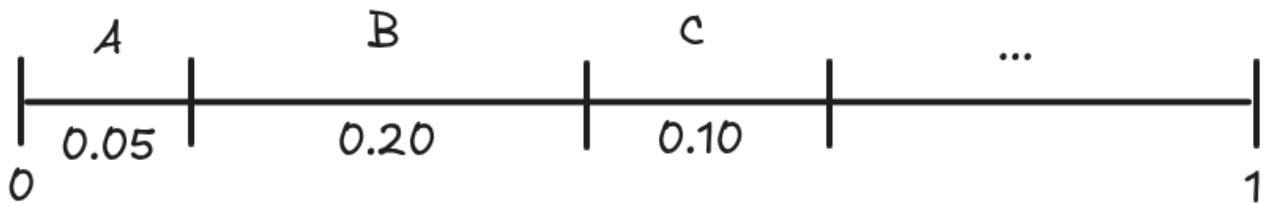
$$V_s = V_{s-1} \cup v_s$$

2. Adicione $M \leq n_0$ arestas (v_s, u) com $u \in V_{s-1}$ com probabilidade:

$$p(u) = \frac{d(u)}{\sum_{w \in V_{s-1}} d(w)} \quad (\text{"preferential attachment": the rich gets richer})$$

(o novo vértice tem grande probabilidade de ligar com quem já tem grau alto)

Faz uma roleta para escolher quem vai ser sorteado com 1 das m arestas:



Quem tem maior probabilidade (hubs), gera um maior intervalo na reta, então sorteia-se um número aleatório $x \in [0,1]$ e verifica-se em qual x caiu.

3. Pare quando $n \gg n_0$ vértices tiverem sido adicionados. Caso contrário, volte para o passo 1.

Obs: após t passos

$$\begin{aligned} |V| &= n_0 + t \\ |E| &= |E_0| + tm \quad (G_0 \text{ pode ser o grafo nulo}) \end{aligned}$$

Teorema: Seja G um grafo gerado pelo algoritmo BA(n, n_0, m). Então:

$$p(k) = \frac{2m(m+1)}{k(k+1)(k+2)} \propto O(k^{-3}) \quad \text{para todo vértice}$$

ou seja, todo grafo gerado pelo algoritmo anterior possui a distribuição de graus como uma lei de potência. (É de fato uma amostra do modelo)

Características:

- i) Distribuição dos graus = lei de potência
- ii) $APL \approx \frac{\log n}{\log(\log \langle k \rangle)}$ (pequeno)
- iii) Coeficiente de aglomeração é baixo
- iv) Modelo mais utilizado para analisar e simular a internet.

Bibliografia

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

Graph Theory and Complex Networks: An Introduction. Maarten van Steen, 2010. Disponível gratuitamente em: <https://www.distributed-systems.net/index.php/books/gtcn/>

Introduction to Algorithms. Thomas H. Cormen, Charles E. Leiserson, Ronald Rivest and Clifford Stein, 4. ed., The MIT Press, 2022.

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

Wilson, Robin. Introduction to Graph Theory, 3 ed., Longman Scientific & Technical, 1985.

Matemática Discreta - Uma Introdução. Edward R. Scheinerman, Thomson Learning, 2003.

Grafos: conceitos, algoritmos e aplicações. M. Goldbarg e E. Goldbarg, Elsevier, 2012.

Algorithmic Design. Jon Kleinberg and Éva Tardos, Addison Wesley, 2005.

Graphs and Applications: An Introductory Approach. Aldous, J. M. & Wilson, R. J., Springer, 2000.

Networks: An Introduction. Newman, M., Oxford University Press, 2010.

Introduction to Graph Theory. Robin J. Wilson, Prentice Hall, England, 1996.

An Introduction to Discrete Mathematics. Steven Roman, Harcourt College Publishers, 1985.

Introduction to Graph Theory. Douglas B. West, 2nd. ed., Prentice-Hall, 2001.

Sobre o autor

Alexandre L. M. Levada é Bacharel em Ciência da Computação pela Universidade Estadual Paulista (UNESP) em 2002. Concluiu seu Mestrado em Ciência da Computação pela Universidade Federal de São Carlos (UFSCar) em 2006 e seu Doutorado em Física Computacional pela Universidade de São Paulo (USP) em 2010. No mesmo ano, ingressou no Departamento de Computação da Universidade Federal de São Carlos. De 2010 a 2024, foi professor assistente no Departamento de Computação, onde lecionou diversas disciplinas de graduação e pós-graduação como lógica matemática, teoria dos grafos, matemática discreta, processamento digital de imagens, análise de sinais e sistemas, reconhecimento de padrões, visão computacional, algoritmos e estruturas de dados 1 e 2, projeto e análise de algoritmos, programação científica e otimização matemática. Desde 2024 é professor associado do mesmo departamento. Seus principais temas de pesquisa incluem redução de ruído em sinais e imagens, reconhecimento de padrões e aprendizado de máquina, com ênfase em algoritmos de classificação baseados em grafos, métodos de redução de dimensionalidade baseados em teoria da informação e aplicação da geometria da informação na análise da dinâmica de campos aleatórios. É autor de um livro, mais de 25 artigos em periódicos e mais de 55 artigos em conferências. Recebeu o prêmio de melhor artigo científico na categoria Inteligência Artificial, Aprendizado de Máquina e Análise de Padrões por um artigo apresentado na 26^a International Conference on Pattern Recognition (ICPR) em 2022. Recebeu uma bolsa de produtividade em pesquisa do CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico) em 2025.

“Experiência não é o que acontece com você; é o que você faz com o que lhe acontece.”
(Aldous Huxley)