UNIP

UNIVERSIDADE PAULISTA

Teoria dos Grafos

Autoras: Profa. Miryam de Moraes
Profa. Vanessa Santos Lessa
Colaboradora: Profa. Larissa Rodrigues Damiani

Professoras conteudistas: Miryam de Moraes / Vanessa Santos Lessa

Miryam de Moraes

Graduada em Engenharia Elétrica (1980), mestra em Engenharia Elétrica (1983) e doutora em Engenharia Elétrica pela Universidade de São Paulo (2006). Exerce docência em nível superior desde 1990. Atualmente, é professora titular da Universidade Paulista (UNIP). Seu doutorado diz respeito ao tratamento de dependências de contexto empregando tecnologia adaptativa. Interessa-se por aplicações da tecnologia adaptativa à engenharia da computação, à análise e processamento de linguagens de programação e linguagens naturais, bem como a processos de aprendizagem e inferências automáticas. É de seu particular interesse o estudo e o ensino das diversas áreas da teoria da computação, a saber: teoria dos grafos, programação de computadores e indução matemática, complexidade de algoritmos e teoria dos autômatos.

Vanessa Santos Lessa

Bacharel em Engenharia da Computação pela Universidade São Judas Tadeu, mestra em Engenharia Elétrica com ênfase em Inteligência Artificial Aplicada à Automação pelo Centro Universitário da Fundação Educacional Inaciana e doutora em Ciências e Aplicações Geoespaciais pelo Instituto Presbiteriano Mackenzie. Coordenadora do curso de Bacharelado em Ciência da Computação na UNIP na modalidade EaD. Atua há mais de 18 anos em cargos técnicos e gerenciais na área de computação.

Dados Internacionais de Catalogação na Publicação (CIP)

M828t Moraes, Miryam de.

Teoria dos Grafos / Miryam de Moraes, Vanessa Santos Lessa. – São Paulo: Editora Sol, 2023.

104 p., il.

Nota: este volume está publicado nos Cadernos de Estudos e Pesquisas da UNIP, Série Didática, ISSN 1517-9230.

1. Grafos. 2. Caminho. 3. Algoritmo. I. Moraes, Miryam de. II. Vanessa Santos Lessa. III. Título.

CDU 519.67

U517.64 – 23

Sumário

Teoria dos Grafos

APRESENTAÇÃO

O matemático suíço Leonhard Euler (1707-1783) tornou-se o pai da teoria dos grafos quando em 1736 resolveu o problema da ponte de Königsberg. Trata-se de um rio que atravessa a cidade e que se bifurca em torno de uma ilha. Como sete pontes atravessam o rio, o problema era decidir se uma pessoa poderia passar por toda a cidade cruzando cada ponte apenas uma vez e retornar ao ponto inicial. Euler identificou que as áreas principais poderiam ser representadas por vértices, também denominados nós, e as pontes, por arestas, também denominadas arcos, que interligavam as áreas. Euler abstraiu o problema e criou regras generalizadas baseadas em nós e relações que se aplicam a qualquer sistema conectado.

O objetivo deste livro-texto é apresentar os conceitos da teoria dos grafos para que você tenha à sua disposição o alicerce para a compreensão das aplicações de teoria dos grafos que compõem atualmente o cenário do mercado profissional, a saber: inteligência artificial, machine learning, deep learning, redes sociais e ciência dos dados.

Vale acrescentar que a linguagem usada é simples e direta, como se houvesse uma conversa entre as autoras e o leitor. Adicionalmente, são inseridas muitas figuras, que auxiliam no entendimento dos tópicos desenvolvidos. Os itens chamados de Observação e de Lembrete são oportunidades para que você solucione eventuais dúvidas. Já os Saiba mais possibilitam que você amplie seus conhecimentos. Há, ainda, muitos exemplos de aplicação resolvidos em detalhes, visando à fixação dos assuntos abordados.

Bons estudos.

INTRODUÇÃO

Muitos tipos de dados são grafos: as interdependências entre sistemas econômicos, imagens detectadas por robôs, distribuição das peças em um tabuleiro de xadrez, recursos de hardware e software a serem gerenciados em um sistema operacional ou em um ambiente distribuído, cidades em um mapa etc.

De fato, muitas instâncias do mundo real podem ser modeladas por grafos ou redes. São exemplos: redes sociais, conexões entre neurônios, links entre websites, links entre elementos de um mundo virtual em realidade virtual e aumentada, relações entre perfis de clientes e produtos em um sistema de recomendação.

O futuro egresso dos cursos da área de informática deverá, portanto, compreender o ferramental da teoria dos grafos empregado nas diferentes áreas do conhecimento, em particular na ciência da computação, para a resolução dos problemas do mundo real.

Este livro-texto tem por objetivo destacar os conceitos da teoria dos grafos. O conteúdo será apresentado em duas unidades.

Na unidade I, apresentamos inicialmente os fundamentos e aplicações da teoria dos grafos.

Na unidade II, abordaremos algoritmos de percurso e suas aplicações.

Esperamos que você tenha uma boa leitura e se sinta motivado a ler e conhecer mais sobre esta disciplina.

Unidade I

1 CONSIDERAÇÕES INICIAIS

A teoria dos grafos é uma área da matemática que estuda as propriedades e as relações entre os elementos de um conjunto, representados por meio de pontos (vértices) e linhas (arestas) que os conectam. Ela teve origem no século XVIII, com o matemático suíço Leonhard Euler, que, em 1736, resolveu o famoso problema das sete pontes de Königsberg.

O problema consistia em determinar se era possível percorrer todas as sete pontes de Königsberg uma única vez, sem passar duas vezes pela mesma ponte. Euler resolveu o problema ao reduzi-lo a um grafo, isto é, um conjunto de pontos e linhas que representavam as pontes e as ilhas da cidade. Ele mostrou que era impossível percorrer todas as pontes uma única vez e, assim, inaugurou a teoria dos grafos.

A figura a seguir representa o problema das sete pontes: na cidade de Königsberg (antiga Prússia) existiam sete pontes que cruzavam o rio Pregel, estabelecendo ligações entre diferentes partes da cidade e as ilhas Kneiphof e Lomse.

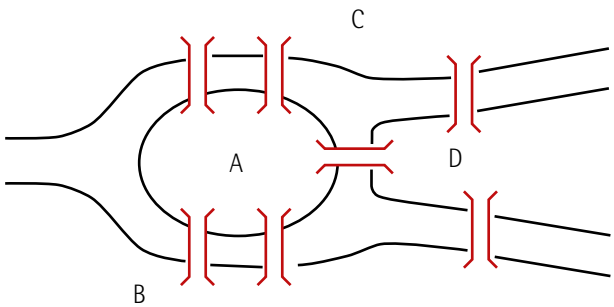


Figura 1

A partir desse momento, a teoria dos grafos se tornou um campo de estudo importante em matemática, informática, física, biologia e outras áreas, com aplicações em problemas como redes de transporte, circuitos elétricos, otimização de rotas, jogos e muitos outros. Desde então, diversos matemáticos e cientistas contribuíram para o desenvolvimento da teoria dos grafos, como Arthur Cayley, William Hamilton, Paul Erdős e muitos outros.

No final do século XIX e início do século XX, a teoria dos grafos foi aplicada em diversos campos, incluindo química, física e engenharia. Por exemplo, o químico alemão August Kekulé usou grafos para representar moléculas orgânicas e suas ligações, enquanto o físico húngaro László Lovász aplicou a teoria dos grafos para entender as propriedades dos sólidos.

A teoria dos grafos continuou a se desenvolver durante todo o século XX, com importantes contribuições de matemáticos como Paul Erdős, Claude Shannon e John Horton Conway. Além disso, a teoria dos grafos se tornou uma ferramenta essencial em muitas áreas, incluindo ciência da computação, redes de comunicação, pesquisa operacional e biologia.

Atualmente, a teoria dos grafos é uma área muito ativa de pesquisa em matemática e em diversos outros campos. Com o advento da era digital, a teoria dos grafos se tornou ainda mais importante, com aplicações em redes sociais, análise de dados, sistemas de recomendação e muitas outras áreas. É uma área fascinante e em constante evolução que continua a contribuir para a compreensão do mundo ao nosso redor.

1.1 Grafos

Nas primeiras anotações do curso Machine Learning with Graphs, Leskovec (2022) destaca que os grafos são definidos como uma linguagem geral para descrever e analisar entidades com relações e interações. Informalmente, um grafo é um conjunto não vazio de nós (vértices) e um conjunto de arcos (arestas), sendo que cada arco conecta dois nós.

Este tópico traz a definição formal de grafos e de dígrafos para contextualizar melhor as aplicações que serão apresentadas em seguida.

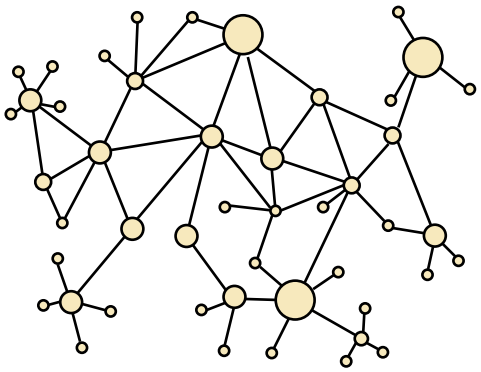
Saiba mais

Os modelos de aprendizado de máquina baseados em grafos podem ser usados para realizar uma variedade de tarefas, como classificação, clusterização, predição e recomendação. Existem várias técnicas de aprendizado de máquina que podem ser usadas com grafos, incluindo redes neurais, modelos probabilísticos e algoritmos de propagação de labels. Para se aprofundar no assunto, leia o capítulo 5 do livro indicado a seguir:

HAYKIN, S. *Redes neurais*: princípios e prática. Porto Alegre: Bookman, 2007.

Podemos observar nas figuras a seguir dois exemplos de representação informal de grafos.

A)



B)

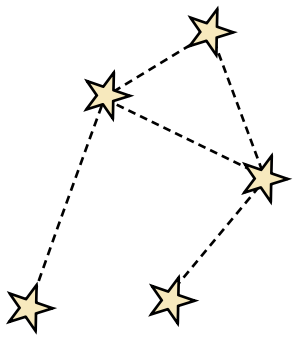


Figura 2

1.1.1 Definição de grafo

Um grafo é uma estrutura matemática composta de um conjunto de vértices (também chamados de nós) e um conjunto de arestas (arcos ou linhas) que conectam esses vértices. Os vértices podem representar entidades ou objetos, enquanto as arestas representam as relações entre essas entidades. Os grafos são frequentemente usados para modelar sistemas complexos, como redes de computadores, sistemas de transporte, relações sociais e muitos outros.

Um grafo é uma tripla ordenada (N, A, g):

- N = um conjunto não vazio de nós (vértices);
- A = um conjunto de arcos (arestas);
- g = uma função que associa cada arco a um par não ordenado (x-y) de nós, denominados extremidades do respectivo arco.

Considere os dois exemplos a seguir.

Exemplo 1

Para o grafo representado graficamente na figura a seguir, tem-se que:

N = {1,2,3}

A = {A1,A2,A3,A4}

g(A1) = 1-2, g(A2) = 2-3, g(A3) = 2-3; g(A4) = 3-3

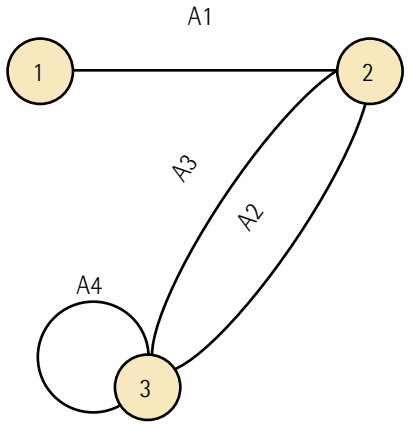


Figura 3 – Exemplo de representação gráfica de um grafo

A figura possui o conjunto de nós 1, 2 e 3 e as arestas A1, A2, A3 e A4. Os nós se relacionam através da função g, onde o argumento da função é a aresta e o resultado da função são as duas extremidades da aresta.

Exemplo 2

Para o grafo representado graficamente na figura a seguir, tem-se:

N = {1,2,3,4,5,6,7,8}

A = {A,B,C,D,E,F,G,H,I,J,K,L}

g(A) = 1-2, g(B) = 2-3, g(C) = 3-4; g(D) = 1-4, e assim por diante

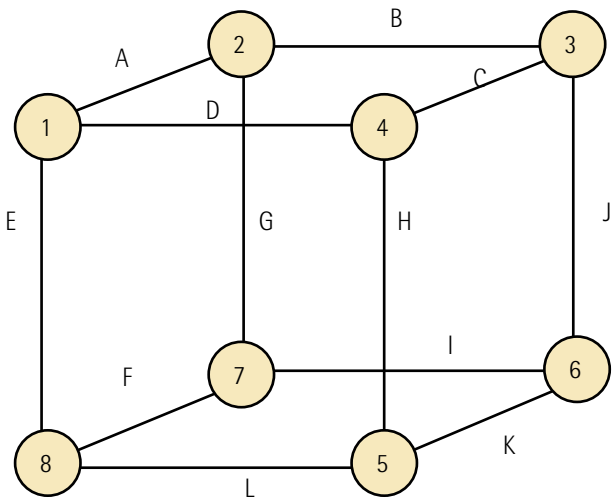


Figura 4

A figura ilustra um gráfico com oito nós e arestas de A até L. A função g da aresta A resulta nos nós 1 e 2; da aresta B, 2 e 3; da aresta C, 3 e 4, e assim sucessivamente.

Lembrete

Os vértices podem representar entidades ou objetos, enquanto as arestas representam as relações entre essas entidades.

1.1.2 Grafo direcionado

Um grafo direcionado, também chamado de dígrafo, é uma estrutura matemática composta de um conjunto de vértices e um conjunto de arcos que conectam esses vértices. Cada arco é um par ordenado de vértices (u, v), onde u é o vértice de origem e v é o vértice de destino. Essa direção indica que a relação entre os vértices é unidirecional, ou seja, há um fluxo ou sentido associado a cada arco.

Os dígrafos podem ser usados para modelar uma ampla variedade de sistemas, como redes de transporte, sistemas de comunicação, circuitos elétricos e muitos outros. Eles são uma ferramenta importante em diversas áreas, incluindo ciência da computação, matemática, engenharia e física.

Em um grafo direcionado, a conexão entre dois vértices não precisa ser recíproca, ou seja, pode haver um arco de u para v, mas não necessariamente um arco de v para u. Além disso, um vértice pode ter arcos que saem dele e arcos que chegam nele, o que permite modelar sistemas com entradas e saídas.

Os dígrafos podem ser expressos graficamente por meio de diagramas que mostram os vértices como pontos e os arcos como setas que apontam da origem para o destino. Essa representação gráfica pode ser útil para visualizar e analisar a estrutura e o comportamento do sistema modelado.

Um grafo direcionado (dígrafo) é uma tripla ordenada (N, A, g):

- N = um conjunto não vazio de nós;
- A = um conjunto de arcos;
- g = uma função associa a cada arco **a** um par ordenado (x,y) de nós, onde x é o ponto inicial e y é o ponto final de **a**.

Em um arco direcionado, cada arco tem um sentido ou orientação.

Considere o exemplo a seguir.

Exemplo 3

Para o grafo direcionado representado graficamente na figura a seguir, tem-se:

N = {1,2,3,4}

A = {A1,A2,A3,A4,A5}

g(A1) = (1,2), g(A2) = (2,3), g(A3) = (3,4); g(A4) = (3,1); g(A5) = (4,3)

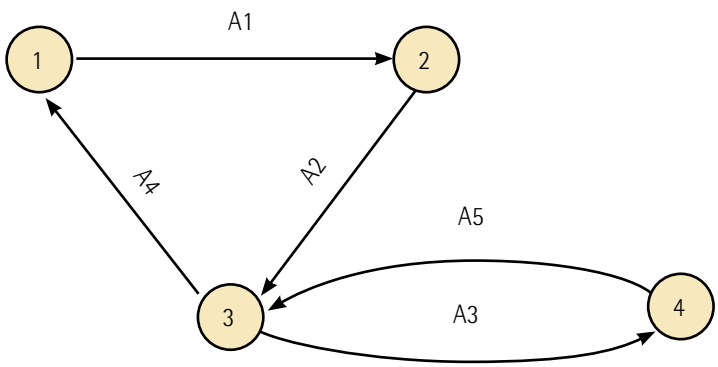


Figura 5

Um grafo direcionado se denomina dígrafo porque as arestas têm uma origem, um sentido e um destino. Por exemplo, na figura anterior, a aresta A1 tem como origem o nó 1 e como destino o nó 2, então cada arco tem um sentido ou uma orientação.

Lembrete

Os dígrafos podem ser usados para modelar uma ampla variedade de sistemas, como redes de transporte, sistemas de comunicação, circuitos elétricos e muitos outros.

1.2 Exemplos de aplicações de grafos

Como vimos, os grafos têm inúmeras aplicações em diversas áreas do conhecimento, como matemática, ciência da computação, engenharia, física, biologia e química. A seguir serão estudadas algumas aplicações de grafos.

1.2.1 Modelagem de um circuito eletrônico através de grafos

A modelagem de um circuito eletrônico através de grafos envolve a representação do circuito como um grafo, em que os componentes eletrônicos são os vértices e as conexões entre eles são as arestas.

Esse tipo de modelagem pode ser muito útil para análise e design de circuitos, permitindo que o circuito seja visualizado de forma clara e simplificada.

Para fazer a modelagem, é necessário seguir alguns passos básicos:

1. Identificar os componentes eletrônicos do circuito: determine todos os componentes eletrônicos do circuito, incluindo resistores, capacitores, indutores, transistores, diodos etc.

2. Identificar as conexões entre os componentes: perceba como os componentes estão conectados entre si, ou seja, quais são os pontos de interconexão do circuito.

3. Atribuir vértices aos componentes: atribua um vértice do grafo a cada componente eletrônico do circuito. Os vértices podem ser representados por símbolos específicos, como círculos ou retângulos.

4. Atribuir arestas às conexões: atribua uma aresta do grafo a cada conexão entre os componentes. As arestas podem ser representadas por linhas que ligam os vértices.

5. Atribuir valores às arestas e vértices, se necessário: se for preciso, os vértices e as arestas podem ser atribuídos com valores específicos, como resistência, capacitância, corrente, tensão etc.

Após a criação do grafo, é possível analisar o circuito de forma mais clara e simplificada. Por exemplo, pode-se identificar caminhos de corrente, calcular resistências equivalentes e analisar as propriedades elétricas do circuito. A modelagem de circuitos eletrônicos através de grafos é uma ferramenta importante para estudantes e profissionais de eletrônica.

Além disso, a modelagem de circuitos eletrônicos através de grafos pode ajudar a identificar problemas, permitindo que sejam feitas correções antes da montagem física do circuito. É possível realizar simulações do circuito utilizando softwares de simulação, que permitem testar o circuito antes da montagem física e fazer ajustes necessários para melhorar seu desempenho.

Os grafos também podem ser usados para expressar circuitos mais complexos, como circuitos integrados, sistemas digitais e redes de computadores. Nesses casos, é possível utilizar grafos direcionados para representar a direção do fluxo de dados ou sinais no circuito.

A modelagem de circuitos eletrônicos através de grafos é uma técnica poderosa e versátil que pode ser aplicada em diversas áreas da eletrônica. Ela pode ajudar a simplificar a análise de circuitos complexos, melhorar o desempenho e a eficiência dos circuitos e facilitar a identificação e correção de problemas no circuito.

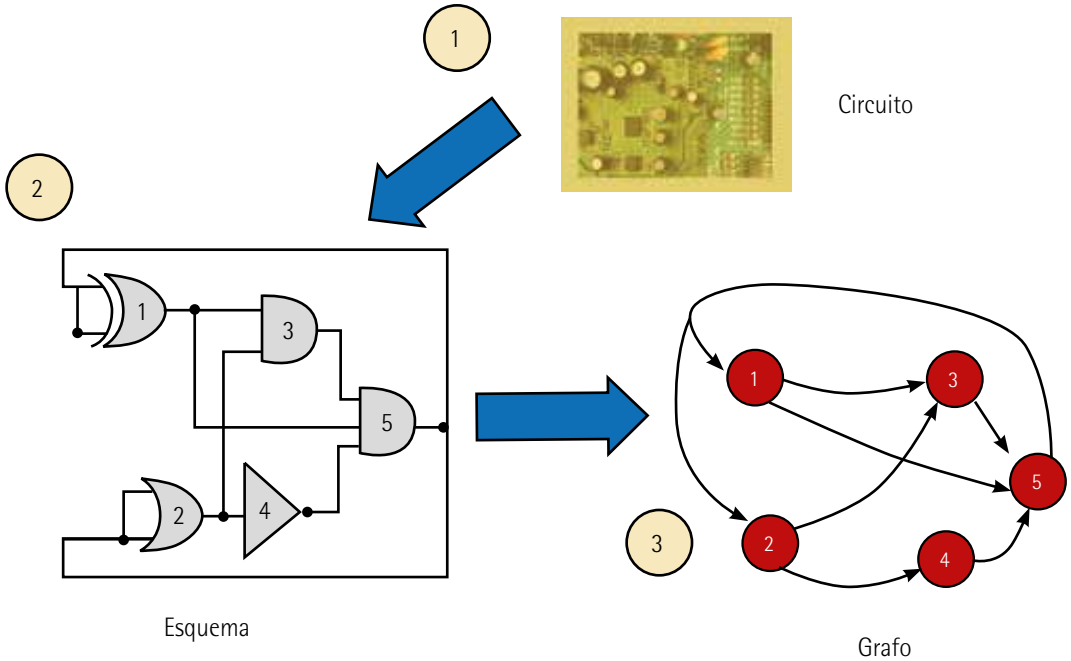


Figura 6 – A formação de um grafo a partir de um circuito eletrônico

Adaptada de: Goldbarg e Goldbarg (2012).

1.2.2 Testabilidade de software

A teoria dos grafos também pode ser usada para avaliar a testabilidade de software, que é a capacidade do software de ser testado de forma eficaz. A ideia é representar o software como um grafo e, em seguida, analisar sua complexidade estrutural para determinar a facilidade ou dificuldade em testá-lo.

Existem várias métricas de testabilidade de software que podem ser calculadas a partir da representação do software como um grafo, e algumas delas incluem:

- **Complexidade ciclomática**: mede a quantidade de caminhos independentes em um grafo de fluxo de controle que representa o software. Quanto maior o número de caminhos independentes, maior a complexidade e mais difícil será testar o software.

- **Grau de coesão**: mede a interdependência entre as unidades de código em um software. Quanto mais interdependentes as unidades de código, mais difícil será testá-las individualmente.

- **Grau de acoplamento**: mede a dependência entre as unidades de código em um software. Quanto maior o grau de acoplamento, mais difícil será testar o software em partes separadas.

- **Profundidade do grafo**: mede a profundidade do grafo que representa o software. Quanto mais profundo o grafo, mais difícil será testar o software, já que haverá mais caminhos a percorrer.

Com base nessas métricas, os desenvolvedores podem avaliar a testabilidade do software e identificar possíveis problemas de qualidade que possam dificultar sua validação. A teoria dos grafos pode ajudar os desenvolvedores a avaliar a complexidade estrutural do software.

Como exemplo da utilização de teoria dos grafos na engenharia de software, vamos destacar um dos métodos para realizar a testabilidade do software implementando o estudo através de grafos. Na figura a seguir, temos um novo arquivo selecionado em alguma aplicação. Esse arquivo selecionado poderá se desdobrar para a geração de uma janela para um documento e um texto desse documento.

Como esse software seria testado? É gerado um novo arquivo; assim, podemos observar que a aresta que associa a janela do documento deve ser exibida em um segundo, algo que deve ser testado; por outro lado, essa janela do documento deve ter um texto que poderá ser configurado. A representação do teste dessas diversas funcionalidades de um software pode ser expressa através de um grafo.

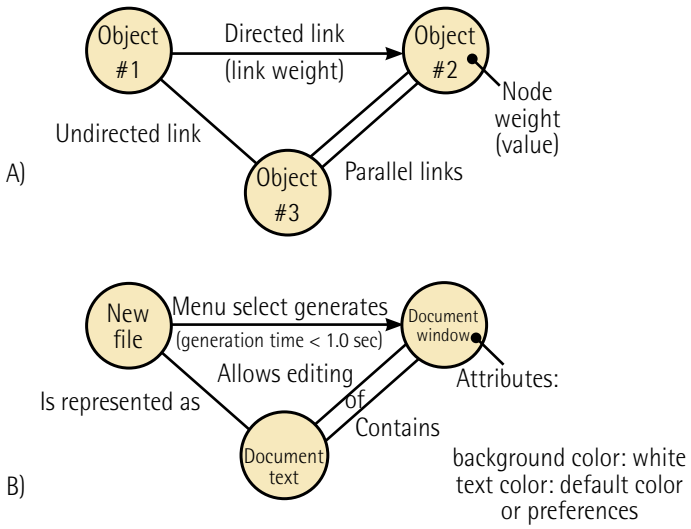


Figura 7

Adaptada de: Pressman e Maxim (2016).

1.2.3 Redes neurais (RNAs)

São modelos computacionais que têm sido usados em diversas áreas de aplicação, como reconhecimento de padrões, classificação e previsão. Em geral, esses modelos são compostos de camadas de neurônios que realizam operações matemáticas para transformar as entradas em saídas.

Recentemente, a teoria dos grafos tem sido utilizada para representar e analisar as RNAs. Em particular, as RNAs podem ser vistas como um tipo de grafo dirigido acíclico (ou grafo direcionado acíclico – DAG, na sigla em inglês), em que cada neurônio é um nó e as conexões entre eles são as arestas. Essa representação permite aplicar técnicas de análise de grafos, como a análise de caminhos mais curtos, a detecção de ciclos e a identificação de comunidades.

Além disso, as técnicas de teoria dos grafos podem ser usadas para otimizar e treinar as RNAs. Por exemplo, o algoritmo de retropropagação é um algoritmo comum de treinamento de RNAs que usa a técnica de gradiente descendente para ajustar os pesos das conexões entre os neurônios. Essa técnica pode ser vista como uma busca pelo caminho mais curto na rede neural, que é um problema de otimização que pode ser resolvido usando algoritmos de teoria dos grafos, como o algoritmo de Dijkstra.

Outra aplicação interessante da teoria dos grafos em RNAs é a identificação de comunidades de neurônios que têm comportamentos semelhantes. Essa abordagem pode ser útil para analisar o funcionamento interno da rede neural e identificar padrões de comportamento que possam ser relevantes para a tarefa em questão.

A teoria dos grafos pode ser uma ferramenta útil para representar e analisar redes neurais artificiais, permitindo que os desenvolvedores identifiquem problemas de otimização, treinamento e comportamento interno da rede neural.

Observe a figura a seguir:

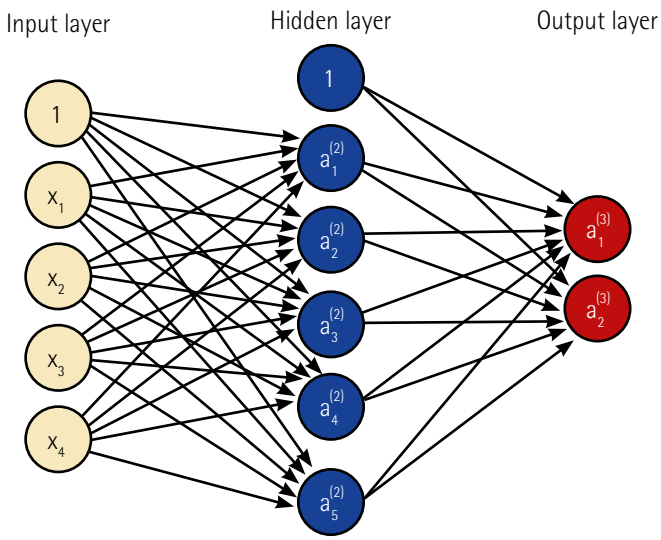


Figura 8 – Ilustração de uma rede neural artificial com quatro entradas e uma entrada de polarização, uma camada oculta com uma unidade de polarização e cinco neurônios, e duas saídas

Adaptada de: Göttlich e Totzeck (2021).

Pode-se modelar uma rede neural como um grafo orientado: cada neurônio é um vértice e as interconexões entre eles correspondem a arcos.

1.2.4 Machine learning (ML)

É uma área da inteligência artificial que utiliza algoritmos para extrair informações e conhecimento a partir de dados. Na teoria dos grafos, algumas técnicas de ML podem ser aplicadas para análise de redes complexas e detecção de padrões.

Uma das principais técnicas utilizadas em ML é a aprendizagem supervisionada, na qual o algoritmo é treinado com um conjunto de dados rotulados. Em teoria dos grafos, isso pode ser utilizado para classificação de grafos ou detecção de anomalias em redes complexas.

Outra técnica comum é a aprendizagem não supervisionada, que permite identificar padrões nos dados sem a necessidade de rótulos pré-definidos. Na teoria dos grafos, isso pode ser útil para detecção de comunidades em redes sociais ou identificação de similaridades entre grafos.

Além disso, existem técnicas de ML que utilizam redes neurais artificiais para modelar e prever o comportamento de sistemas complexos. Em teoria dos grafos, isso pode ser aplicado para previsão de evolução de redes dinâmicas ou para análise de interações entre diferentes elementos de uma rede complexa.

A aplicação de técnicas de ML em teoria dos grafos permite extrair informações valiosas a partir de dados complexos, possibilitando a detecção de padrões, previsão de comportamentos e análise de redes de grande escala.

Por que usar grafos em aprendizado de máquina? Leskovec (2002) responde que grafos constituem uma linguagem geral para descrição e análise de entidades que se inter-relacionam e/ou interagem.

A figura a seguir ilustra o uso de grafos, e em seus nós agregamos informações de seus adjacentes (nós vizinhos) usando redes neurais.

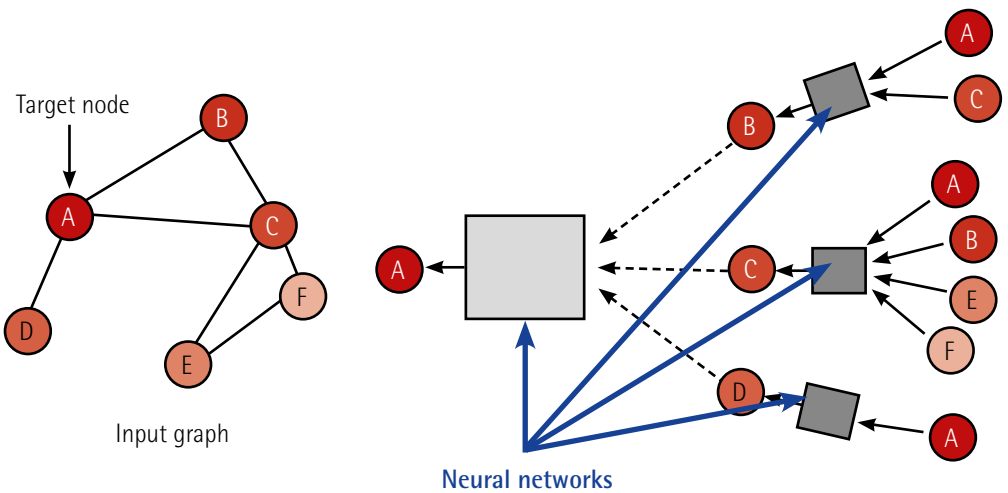


Figura 9 – Redes neurais para grafos

Adaptada de: Hamilton, Ying e Leskovec (2017).

Conforme Calaça (2020), as aplicações das redes neurais para grafos são diversas, desde a composição e classificação de estruturas moleculares (na geração de novos medicamentos, por exemplo) até o reconhecimento de padrões em redes sociais.

1.2.5 Sistemas operacionais

Em sistemas operacionais, a teoria dos grafos é utilizada para modelar e analisar diversas estruturas e processos que ocorrem em um sistema computacional. A seguir são elencadas algumas das principais aplicações da teoria dos grafos em sistemas operacionais:

- **Gerenciamento de processos**: a execução de processos em um sistema operacional pode ser modelada como um DAG, no qual cada nó representa um processo e as arestas expressam as dependências entre eles. Essa representação permite a análise do desempenho do sistema, identificação de gargalos e otimização do tempo de execução.

- **Gerenciamento de memória**: também pode ser modelado como um grafo. Nesse caso, os nós representam as regiões de memória e as arestas expressam as dependências entre elas. Essa modelagem permite a otimização do uso da memória e a prevenção de vazamentos de memória.

- **Gerenciamento de rede**: a comunicação entre processos em uma rede pode ser modelada como um grafo, no qual cada nó representa um processo e as arestas expressam as conexões de rede entre eles. Essa modelagem permite a análise do tráfego de rede, identificação de gargalos e otimização da transferência de dados.

- **Gerenciamento de arquivos**: a organização dos arquivos em um sistema de arquivos pode ser modelada como uma estrutura de árvore. Essa modelagem permite a análise da estrutura dos arquivos, identificação de arquivos duplicados e otimização do tempo de acesso aos arquivos.

- **Gerenciamento de recursos**: a alocação de recursos em sistemas operacionais pode ser modelada como um grafo bipartido, no qual os nós de um conjunto representam os processos e os nós de outro conjunto expressam os recursos disponíveis. Essa modelagem permite a otimização da alocação de recursos e a prevenção de conflitos entre processos.

Em resumo, a teoria dos grafos é uma ferramenta importante para o desenvolvimento e a otimização de sistemas operacionais, permitindo a análise de diversas estruturas e processos críticos para o desempenho do sistema.

A figura a seguir destaca um exemplo de grafo de alocação de recursos com impasse, o qual permite identificar claramente a dependência cíclica entre tarefas e recursos no ciclo.

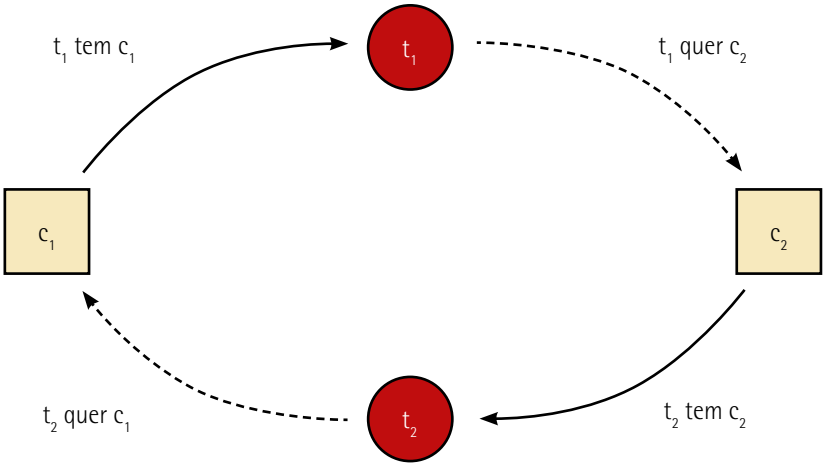


Figura 10 – Gráfico de alocação de recursos com impasses

Adaptada de: Muchalski e Maziero (2014).

1.2.6 Grafo de visibilidade

É um conceito importante em teoria dos grafos e é utilizado em muitas aplicações, como robótica, computação gráfica, jogos e geografia computacional. Ele é um grafo que representa a relação de visibilidade entre pontos em um ambiente bidimensional.

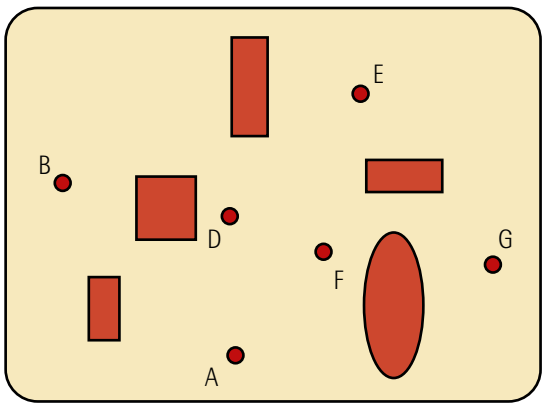
Em um ambiente bidimensional, um ponto é visível a outro se não há obstáculos no caminho entre eles. O grafo de visibilidade é construído a partir dos pontos que representam os vértices e as linhas que expressam as arestas. Cada aresta do grafo conecta dois pontos que são visíveis um ao outro.

Existem diferentes algoritmos para construir o grafo de visibilidade, mas em geral eles se baseiam em técnicas de varredura de linha ou de varredura angular. A ideia é percorrer todos os pontos do ambiente e determinar quais pontos são visíveis uns aos outros. Esse processo pode ser feito de forma eficiente usando algoritmos como o de Bentley-Ottmann, que utiliza uma estrutura de dados de linha de varredura para detectar interseções entre linhas em tempo linear.

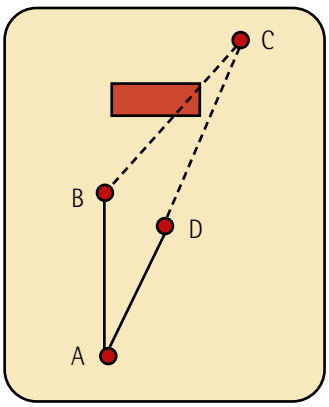
O grafo de visibilidade tem várias aplicações importantes em computação gráfica. Por exemplo, ele pode ser usado para determinar se um objeto está visível ou oculto em uma cena tridimensional, ou para determinar quais objetos são afetados pelas sombras em uma cena iluminada. Além disso, ele pode ser usado para otimizar a renderização de gráficos em tempo real, permitindo que apenas os objetos visíveis sejam renderizados.

Como dito, o grafo de visibilidade é uma importante ferramenta da teoria dos grafos que tem muitas aplicações práticas em computação gráfica, robótica, jogos e geografia computacional. Ele é usado para determinar quais pontos são visíveis uns aos outros em um ambiente bidimensional e pode ser construído de forma eficiente usando técnicas de varredura de linha ou varredura angular.

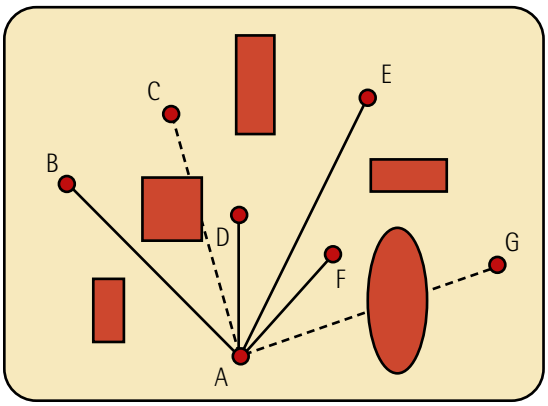
Observe a figura a seguir. Goldbarg e Goldbarg (2012) explicam que uma utilização do modelo de grafos pode ser encontrada no tratamento da representação da visibilidade, com aplicações em robótica e geometria. Considere um grafo G = (N,M) no qual os vértices representam posições de observação e as arestas (i,j) marcam se o vértice i enxerga o vértice j.



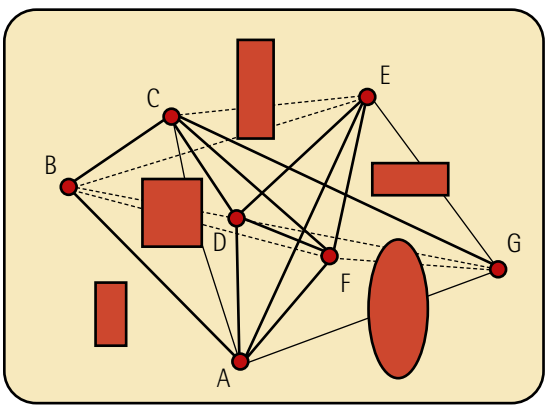
(1) Cenário



(2) Ponto obstáculo



(3) Visibilidade do ponto A



(4) Grafo de visibilidade

Figura 11

Adaptada de: Goldbarg e Goldbarg (2012).

2 GRAFOS E ÁRVORES

Em teoria dos grafos, um grafo é uma coleção de pontos (ou vértices) que podem estar conectados por linhas (ou arestas). Uma árvore é um tipo especial de grafo que não contém ciclos e está conectado.

Um grafo pode ser transformado em uma árvore ao remover algumas de suas arestas, de tal maneira que não haja mais ciclos no grafo resultante. Esse processo é chamado de árvore geradora. Se o grafo é conexo, uma árvore geradora é uma árvore que conecta todos os vértices do grafo.

As árvores têm propriedades importantes que as tornam úteis em muitas aplicações. Por exemplo, as árvores são usadas para representar a estrutura hierárquica em muitos sistemas, como os sistemas de arquivos em um computador. As árvores também são usadas em algoritmos de busca em largura e em profundidade, e em algoritmos de caminho mínimo, como o algoritmo de Dijkstra.

Outra importante aplicação das árvores em teoria dos grafos é a árvore de busca binária. Essa é uma árvore binária em que cada nó possui no máximo dois filhos, e cujos valores armazenados nos nós da árvore são organizados de forma a permitir a rápida busca de um valor específico na árvore. A árvore de busca binária é amplamente utilizada em computação para implementar estruturas de dados, como árvores de pesquisa e mapas.

Árvores são um tipo especial de grafo acíclico, ou seja, um grafo sem ciclos. Uma árvore é formada por um conjunto de vértices e um conjunto de arestas que conectam os vértices. A árvore é uma estrutura hierárquica, em que um vértice é designado como a raiz da árvore e todos os outros vértices são conectados a ela através de caminhos únicos. Cada vértice da árvore, exceto a raiz, é conectado a um vértice pai, e cada vértice pode ter zero ou mais vértices filhos.

As árvores são utilizadas em teoria dos grafos e em muitas aplicações práticas, como estruturas de dados, algoritmos de busca e indexação, e em redes de computadores para gerenciamento de recursos e comunicação.

Existem vários tipos de árvores, cada uma com suas próprias características e propriedades. Algumas das árvores mais comuns são:

- **árvores binárias**: cada vértice tem no máximo dois filhos;
- **árvores de busca binária**: cada vértice tem um valor associado e a chave de cada filho à esquerda é menor do que a chave do pai, enquanto a chave de cada filho à direita é maior ou igual à chave do pai;
- **árvores balanceadas**: garantem que a altura da árvore seja proporcional ao logaritmo do número de vértices;
- **árvores B**: são árvores balanceadas em disco, usadas em bancos de dados;
- **árvores Trie**: usadas em processamento de texto e pesquisa de palavras-chave.

As árvores são muito importantes em teoria dos grafos, pois muitos problemas podem ser resolvidos de forma mais eficiente em árvores do que em grafos gerais. Alguns exemplos de problemas que podem ser resolvidos de forma mais eficiente em árvores são:

- busca em largura e profundidade;
- árvores de busca binária e algoritmos de ordenação;
- árvores de decisão em aprendizado de máquina;
- algoritmos de codificação e decodificação de Huffman em compressão de dados.

Portanto, entender as propriedades e características das árvores é fundamental para a compreensão de muitos conceitos importantes em teoria dos grafos e sua aplicação em várias áreas do conhecimento.

2.1 Terminologia da teoria dos grafos

Observe a seguir as definições:

- Dois nós em um grafo são ditos **adjacentes** se ambos são extremidades de algum arco.
- Um **laço** em um arco é um arco com extremidades n-n para algum nó n.
- Dois arcos com as mesmas extremidades são ditos **arcos paralelos**.
- Um **grafo simples** é um arco que não tem laços nem arcos paralelos.
- Um **nó isolado** é um nó que não é adjacente a nenhum outro.
- O **grau de um nó** é o número de arcos que terminam naquele nó.

Exemplo 4

Na figura a seguir, tem-se que:

- O nó 2 é adjacente aos nós 1 e 3.
- A2 e A3 são arcos paralelos e A4 é um laço.
- O grafo não é simples.
- grau(1) = 1; grau(2) = 3; grau(3) = 3.

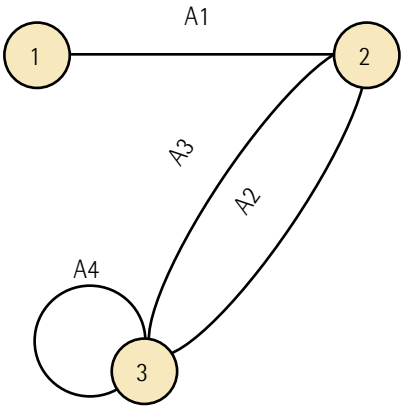


Figura 12 – Exemplo de um grafo

Conforme a figura, o grau do nó 1 é 1, o grau do nó 2 é 3 e o grau do nó 3 é 3, porque existem três arestas incidindo no nó 2 e igualmente no nó 3, onde há um laço e uma aresta incidindo, e duas arestas, A2 e A3.

Exemplo 5

Na figura a seguir, tem-se que:

- O nó a é adjacente aos nós b, h e d.
- O grafo é simples.
- O grau de todos os nós é 3.

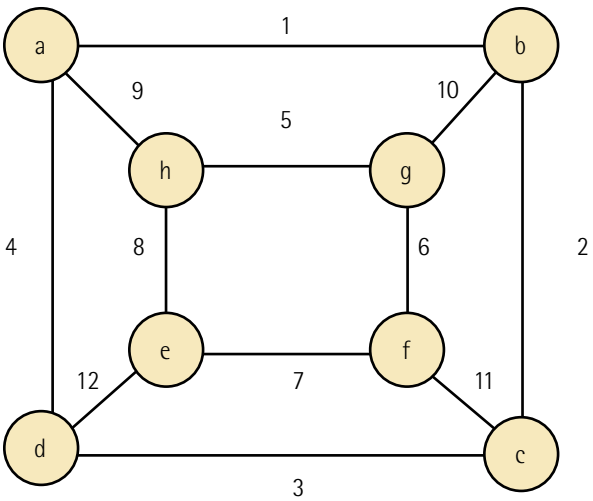


Figura 13 – Representação gráfica de um grafo

Um grafo completo é um grafo no qual dois nós distintos quaisquer são adjacentes. Nesse caso, g é uma função sobrejetora e todo par x-y de nós distintos é a imagem, sob g, de algum arco, mas

não há necessidade de se ter um laço em cada nó. Portanto, pares de forma x-x podem não ter uma imagem inversa.

O grafo simples completo com n vértices é denotado por Kn.

Exemplo 6

Para n = 1, n = 2 e n = 5, têm-se os grafos completos denominados K1, K2 e K5, respectivamente. Observe a figura a seguir.

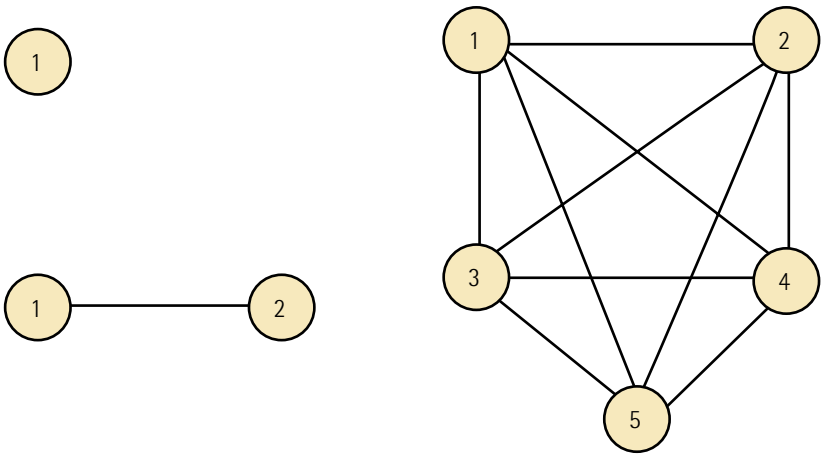


Figura 14 – Representação gráfica dos grafos K1, K2 e K5

Um subgrafo de um grafo consiste em um conjunto de nós e um conjunto de arcos que são subconjuntos do conjunto original de nós e arcos, respectivamente, nos quais as extremidades de um arco têm que ser as mesmas que no grafo original.

Exemplo 7

Na figura a seguir, G2 é subgrafo de G1:

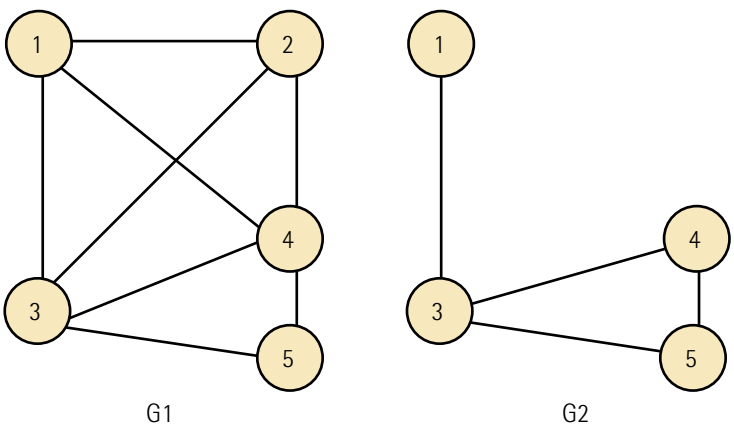


Figura 15 – Representação de um grafo e subgrafo respectivo

- Um caminho do nó n0 para o nó nK é uma sequência n0, a0, n1, a1, ..., nk-1, ak, nk.
- O comprimento de um caminho é o número de arcos que ele contém.
- Um grafo é conexo se existe um caminho de qualquer nó para qualquer outro.
- Um ciclo em um grafo é um caminho de algum nó n0 para ele mesmo, tal que nenhum arco aparece mais de uma vez, n0 é o único nó que aparece mais de uma vez e n0 aparece apenas nas extremidades.
- Um grafo sem ciclos é dito acíclico.

Exemplo 8

Na figura a seguir, tem-se que:

- G1 é acíclico e G2 apresenta ciclo.
- G1 e G2 são conexos.
- Para ambos os grafos, tem-se como exemplo os caminhos de comprimento 3, a saber: 2A1B3C4 em G1 e G2; 4C3D2A1 em G2.
- Ciclo em G2: 3B1A2D3.

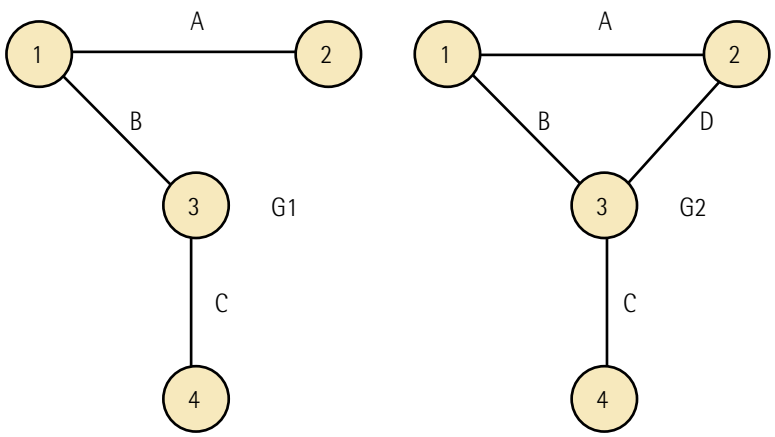


Figura 16 – Representação gráfica de dois grafos

Um caminho de um nó n0 para um nó nk em um grafo direcionado é uma sequência. Nela, para cada ni, ni é o ponto inicial, e ni+1 é o ponto final do arco ai. Se existe um caminho de n0 para nk, diz-se que nk é acessível de n0. A definição de ciclo também pode ser estendida para grafos direcionados.

Exemplo 9

Na figura a seguir, tem-se que:

- **Exemplo de caminho**: 1 A1 2.
- **Exemplo de ciclo**: 3 A3 4 A5 3.

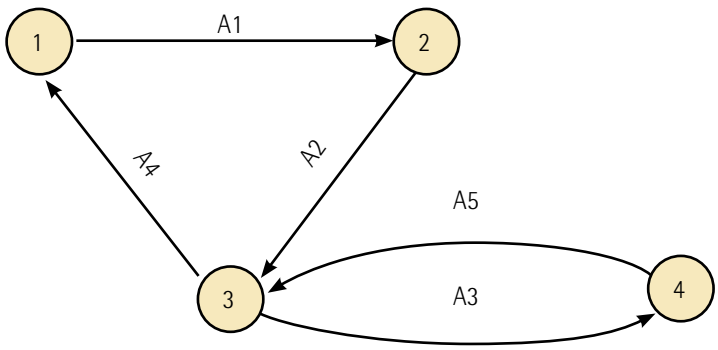


Figura 17 – Representação gráfica de um grafo direcionado

Um grafo é um grafo bipartido completo se seus nós podem ser divididos em dois conjuntos disjuntos não vazios N1 e N2, tais que dois nós são adjacentes se, e somente se, um deles pertence a N1 e o outro pertence a N2. Se |N1| = m e |N2| = n, um tal grafo é denotado por Km,n.

Exemplo 10

Conforme a figura a seguir, tem-se que:

N1= {1,2,3}, portanto |N1|=3

N2 = {4,5}, portanto |N2|=2

N1 e N2 são disjuntos, portanto o grafo é bipartido completo.

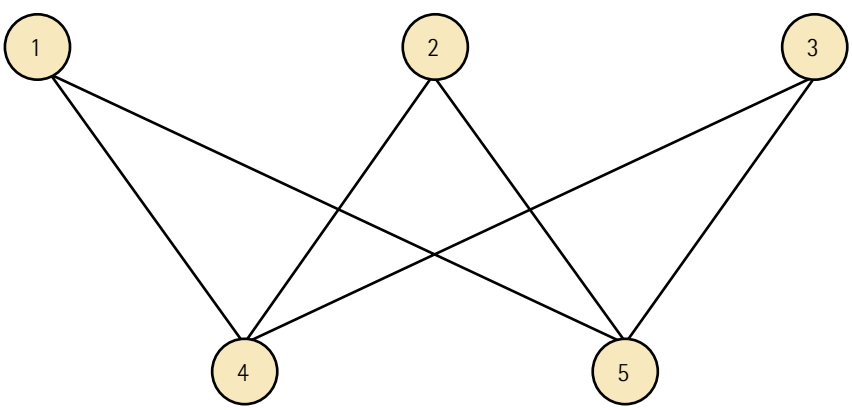


Figura 18 – Representação gráfica do grafo $K_{3,2}$

2.2 Grafos isomorfos

Trata-se de grafos que têm a mesma estrutura, mas podem ser desenhados de forma diferente. Em outras palavras, dois grafos são isomorfos se eles têm o mesmo número de vértices, o mesmo número de arestas e a mesma conectividade, mas a disposição dos vértices e das arestas pode ser diferente.

Dois grafos, (N1, A1, g1) e (N2, A2, g2), são isomorfos se existem bijeções:

f1: N1 → N2 e f2: A1 → A2 tais que, para cada arco a ∈ A1, g1(a) = x-y se, e somente se, g2[f2(a)] = f1(x) – f1(y).

Observação

Uma bijeção ou uma função bijetora é uma função que é sobrejetora e é uma função injetora, ou seja, todos os nós de N1 estão associados por uma função F1 a todos os nós N2; se o nó x de N1 tiver uma função F1 de x a um nó y em N2, ela é única e não vale para o outro nó.

Exemplo 11

Os grafos representados na figura a seguir são isomorfos, conforme as bijeções f1 e f2. De fato, tem-se:

f1: a → 4 ; f1: b → 5; f1: c → 3 ; f1: d → 1; f1: e → 2

f2: x4 → y1; f2: x3 → y7; f2: x2 → y3; f2: x1 → y4; f2: x6 → y2; f2: x5 → y5;

f2: x7 → y6

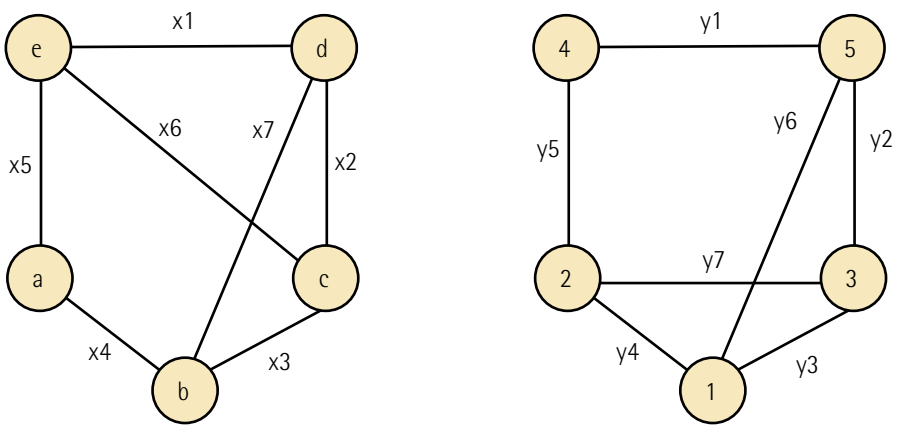


Figura 19 – Grafos isomorfos

Exemplo 12

Os grafos representados na figura a seguir são isomorfos, conforme as bijeções f1 e f2. De fato, tem-se:

Os grafos ao lado são isomorfos.

Bijeção entre os nós:

f1: 1 → a; 2 → b; 3 → c; 4 → d; 5 → e; 6 → f; g → 7; h → 8

Bijeção entre as arestas:

f2: A → 1; B → 2; C → 3; D → 4; E → 9; F → 5; G → 10; H → 12; I → 6; J → 11;

K → 7; L → 8

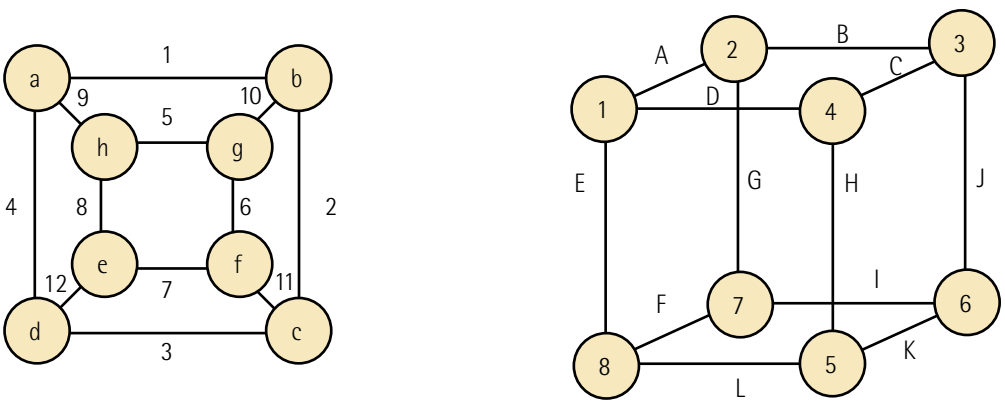


Figura 20 – Dois grafos isomorfos

2.3 Grafos planares

Um grafo planar é um grafo que pode ser representado de modo que seus arcos se intersectam apenas em nós.

Um nó completo (também chamado de grafo completo) é um subgrafo de um grafo em que todos os vértices estão conectados entre si. Em outras palavras, em um nó completo de n vértices, cada par de vértices é adjacente por uma aresta.

Um nó completo com n vértices é denotado como K_n, onde n é o número de vértices. Por exemplo, K_3 é um nó completo com três vértices, e K_4 é um nó completo com quatro vértices. O número de arestas em um nó completo K_n pode ser calculado usando a fórmula:

E(K_n) = (n * (n-1)) / 2

Isso ocorre porque cada vértice em um nó completo está conectado a todos os outros vértices, exceto ele mesmo, e cada par de vértices é contado duas vezes na contagem total de arestas (uma vez para cada extremidade da aresta). Portanto, o número total de arestas em um nó completo com n vértices é a combinação de n escolhendo 2, que é o mesmo que (n * (n-1)) / 2.

Teorema 1

Para um grafo planar simples e conexo com n nós e a arcos:

Se a representação planar divide o plano em r regiões, então: $n - a + r = 2$.

Teorema 2

Para um grafo planar simples e conexo com n nós e a arcos:

Se $n \geq 3$, então $a \leq 3n - 6$.

Teorema 3

Para um grafo planar simples e conexo com n nós e a arcos:

Se $n \geq 3$ e se não existem ciclos de comprimento 3, então $a \leq 2n - 4$.

Exemplo 13

Na figura a seguir, tem-se que:

- Trata-se do grafo completo K4, representado graficamente como G1 e G2, que são isomorfos e planares.
- Para o grafo K4 vale o teorema 1, pois $n = 4$ e $a = 6$. Em G2 é evidente que o número de regiões é $r = 4$. A região externa também é contabilizada.
- Tem-se que $a \leq 3n - 6$, portanto, vale o teorema 2, ou seja, $6 \leq 6$.
- Por outro lado, K4 apresenta ciclo de comprimento 3, portanto, não vale o teorema 3.

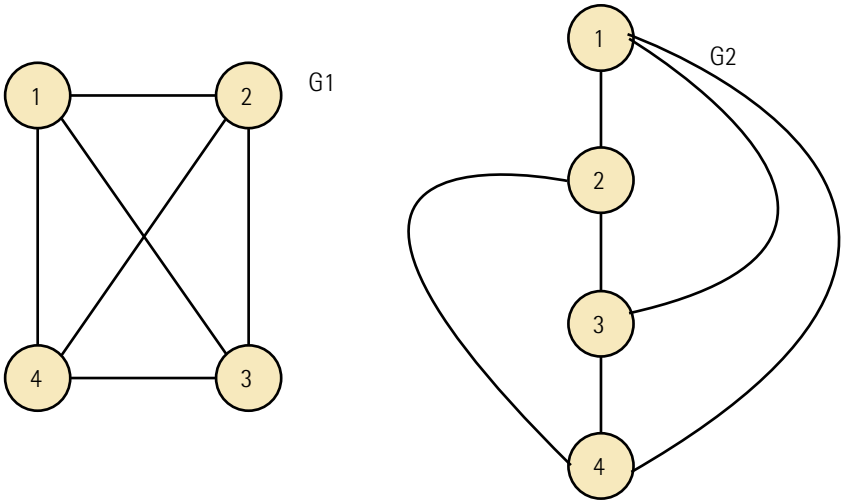


Figura 21 – K4

Exemplo 14

O grafo K5 é não planar. A figura 22 diz respeito à representação gráfica do grafo K5, e na figura 23 é possível inferir que na elaboração do grafo não seria possível concluir sua representação gráfica sem que houvesse cruzamento de duas arestas em um ponto, no plano, distinto do vértice.

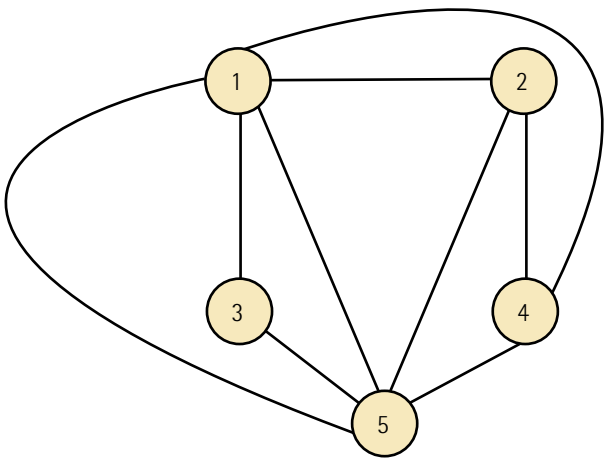


Figura 22 – Grafo K5

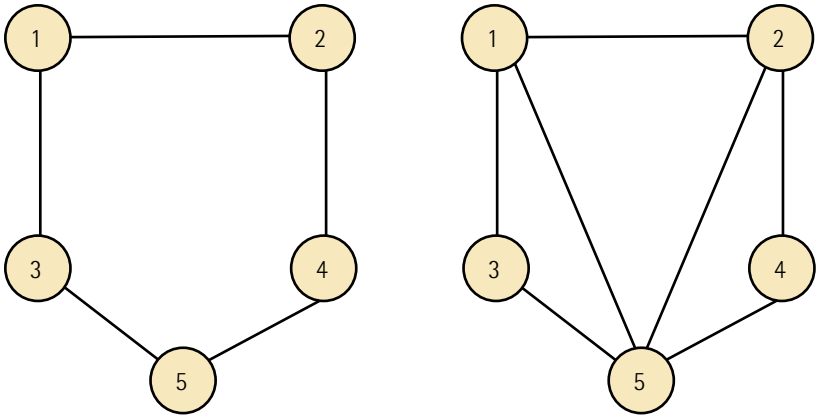


Figura 23 – Duas etapas de elaboração do grafo K5

Dois grafos são ditos **homeomorfos** se ambos podem ser obtidos do mesmo grafo por uma sequência de subdivisões elementares, nas quais um único arco x-y é substituído por dois novos arcos, x-v e v-y.

Exemplo 15

Na figura a seguir, observa-se primeiro a representação gráfica do grafo bipartido completo K33 e, depois, a expressão de um grafo homeomorfo a K33. De fato, o arco 1-4 é substituído no grafo homeomorfo pelos arcos 1-7 e 7-4.

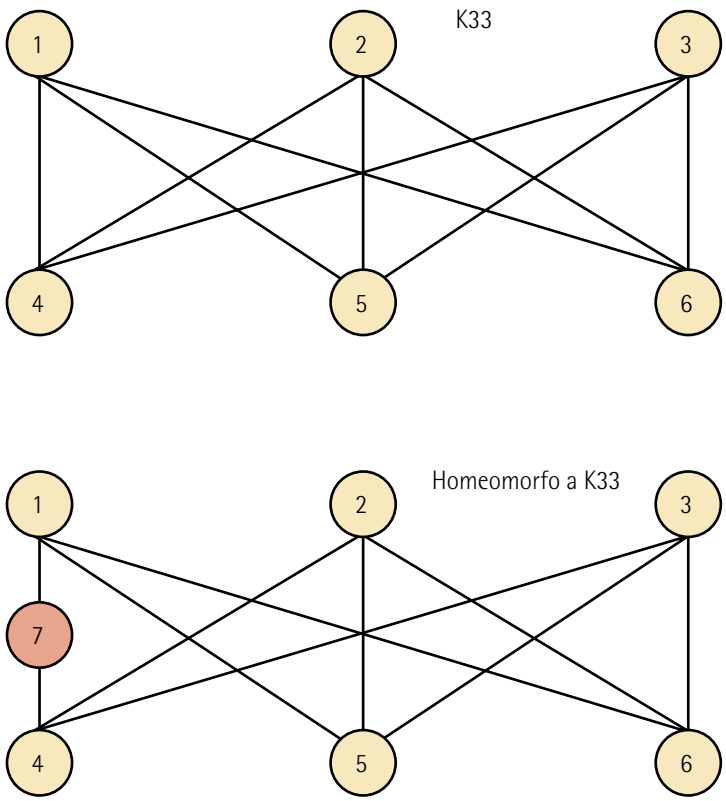


Figura 24 – Representação gráfica do grafo K33 e de um grafo homeomorfo

Nesse contexto, destaca-se o **teorema de Kuratowski**, um resultado fundamental da teoria dos grafos que estabelece uma condição necessária e suficiente para um grafo ser planar. Ele foi enunciado pelo matemático polonês Kazimierz Kuratowski em 1930. O teorema afirma que um grafo é planar se e somente se ele não contém um subgrafo que seja uma contração de K5 (um nó completo com cinco vértices) ou K3,3 (dois conjuntos de três vértices conectados por arestas alternadas). Em outras palavras, se um grafo contém um subgrafo que pode ser transformado em um nó completo K5 ou em um nó completo bipartido K3,3 por meio da compressão de arestas, então o grafo não é planar. O teorema de Kuratowski é vital para a teoria dos grafos, pois permite determinar de forma eficiente se um grafo é planar ou não.

2.4 Representação de grafos no computador

Essa é uma área importante da teoria dos grafos, e tem aplicação em diversas áreas. Existem diversas formas de representar grafos no computador, cada uma com suas vantagens e desvantagens.

Uma das formas mais comuns de representar um grafo no computador é usando uma matriz de adjacência. Nela, um grafo com n vértices é expresso por uma matriz n x n, onde cada elemento aij da matriz representa uma aresta entre os vértices i e j. Se a aresta existe, o valor de aij é 1; caso contrário, o valor é 0. Essa representação é útil para grafos densos, ou seja, grafos com muitas arestas, porque o tempo de acesso a uma aresta é constante.

Outra forma de ilustrar um grafo no computador é usando uma lista de adjacência. Nela, cada vértice do grafo é representado por uma lista que contém os vértices adjacentes a ele. Essa representação é útil para grafos esparsos, ou seja, grafos com poucas arestas, porque o espaço necessário para armazenar as listas é proporcional ao número de arestas.

Existem ainda outras formas de representação de grafos no computador, como a matriz de incidência, que ilustra cada aresta por uma coluna da matriz, e a representação por objetos, em que cada vértice e aresta do grafo é expresso por um objeto.

A representação de grafos no computador é útil para diversas aplicações, e é importante escolher a forma mais adequada para cada problema, levando em consideração as características do grafo e as necessidades da aplicação. Vamos exemplificar a seguir as duas formas mais utilizadas: matriz de adjacência e lista de adjacência.

Matriz de adjacência

Suponha que um grafo tem n nós numerados n1, n2, ..., nn para sua identificação. Uma vez que eles sejam ordenados, pode-se formar uma matriz quadrada n x n, em que o elemento i,j é o número de arcos entre os nós ni e nj. Essa matriz é chamada de matriz de adjacência.

A matriz de um grafo não direcionado é simétrica.

Exemplo 16

Na figura a seguir, $M = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 2 & 1 \end{bmatrix}$ é a matriz de adjacência.

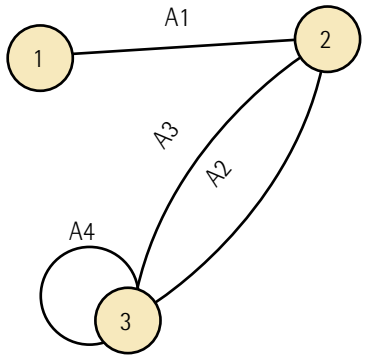


Figura 25 – Grafo não direcionado

Lista de adjacência

Um grafo com relativamente poucos arcos pode ser representado de modo mais eficiente armazenando-se apenas os elementos não nulos da matriz de adjacência, ou seja, através de uma lista de adjacência. Esta contém um arranjo de ponteiros, um para cada nó. Cada ponteiro do arranjo aponta para uma lista encadeada de nós adjacentes.

Exemplo 17

Seja a matriz de adjacência do exemplo anterior $M = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 2 & 2 \\ 0 & 2 & 1 \end{bmatrix}$, a lista de adjacência correspondente é apresentada na figura a seguir.

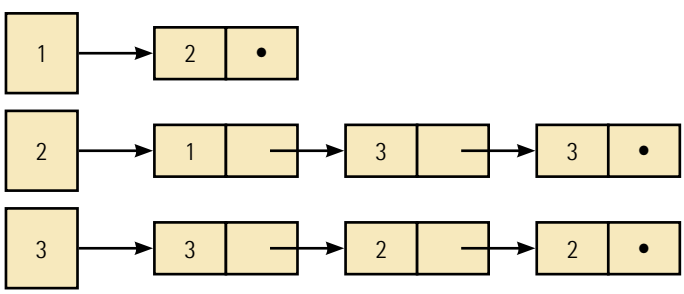


Figura 26 – Lista de adjacência

Na **representação de grafos no computador para dígrafos**, os conceitos de matriz e lista de adjacência também se aplicam.

Exemplo 18

Como exemplo, veja o dígrafo representado na figura a seguir:

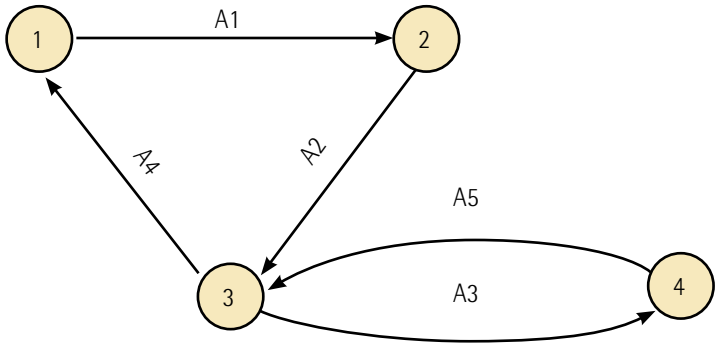


Figura 27 – Exemplo de um dígrafo

A matriz de adjacência correspondente é $M = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

Naturalmente, para dígrafos, a matriz de adjacência não é simétrica.

A lista de adjacência correspondente é:

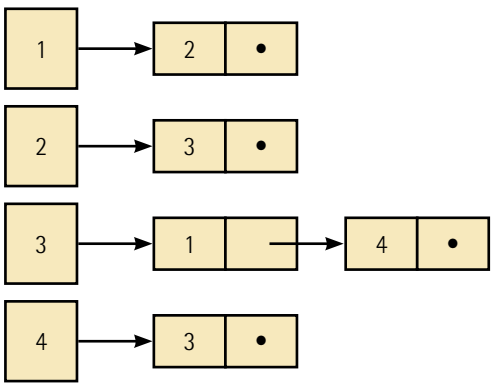


Figura 28 – Lista de adjacência de um dígrafo

2.5 Árvores

Uma árvore é um tipo especial de grafo conexo e acíclico, formado por um conjunto de vértices e um conjunto de arestas que conectam os vértices entre si. Cada vértice da árvore, com exceção de um vértice especial chamado raiz, está conectado a exatamente uma aresta que vem de um vértice pai.

Uma árvore é caracterizada por ter um caminho único entre dois vértices quaisquer. Essa propriedade é conhecida como unicidade de caminho. Além disso, uma árvore tem exatamente n-1 arestas, em que n é o número de vértices.

As árvores são amplamente utilizadas em teoria dos grafos e em diversas áreas da ciência da computação, como em algoritmos de busca, estruturas de dados, redes de computadores e em inteligência artificial.

Uma árvore também pode ser vista como uma hierarquia, em que o vértice raiz representa o nível mais alto da hierarquia e os vértices adjacentes expressam os níveis inferiores. Por essa razão, as árvores são frequentemente usadas para ilustrar hierarquias em sistemas de computação.

Existem diversas propriedades interessantes que caracterizam as árvores em teoria dos grafos. Algumas dessas propriedades incluem:

- uma árvore com n vértices tem exatamente n-1 arestas;
- qualquer subgrafo de uma árvore é uma árvore;
- adicionar uma aresta a uma árvore cria um ciclo;
- remover uma aresta de uma árvore cria dois componentes conexos;
- se uma árvore tem ao menos dois vértices, então ela tem ao menos duas folhas (vértices de grau 1).

As árvores são utilizadas em algoritmos de busca e otimização, como a busca em profundidade, a busca em largura e o algoritmo de Kruskal, para encontrar a árvore geradora mínima de um grafo. Além disso, as árvores são frequentemente aplicadas em estruturas de dados, como as árvores binárias de busca, as árvores AVL e as árvores de decisão.

Saiba mais

Para saber mais sobre árvores, árvores binárias de busca e árvores balanceadas, leia os capítulos 3, 4 e 5 do livro indicado a seguir:

SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de dados e seus algoritmos*. 3. ed. Rio de Janeiro: LTC, 2010.

Observe a seguir alguns pontos importantes:

- uma árvore é um grafo conexo acíclico com um nó especial, denominado raiz da árvore;
- a profundidade de um nó em uma árvore é o comprimento do caminho da raiz ao nó, e a raiz tem profundidade 0;
- a profundidade (altura) de uma árvore é a maior profundidade dos nós na árvore;

- um nó sem filhos é chamado de folha da árvore, e todos os nós que não são folhas são nós internos;

- uma floresta é um grafo acíclico (não necessariamente conexo); logo, uma floresta é uma coleção de árvores disjuntas.

Exemplo 19

Veja a árvore representada na figura a seguir. A árvore tem como raiz a e tem os nós b, c, d, e, f, g, h e i. A altura dessa árvore é 3, pois na raiz temos a profundidade 0 (o comprimento da raiz para raiz é zero); da raiz para os nós b e c temos altura e profundidade com uma aresta com profundidade 1; da raiz para os nós d, e, f e g temos altura e profundidade 2; da raiz até os nós h e i, temos altura e profundidade igual a 3. Portanto, a altura da árvore é 3.

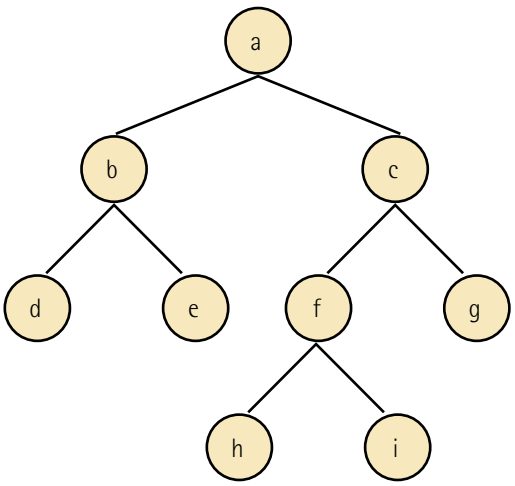


Figura 29 – Árvore com altura e profundidade iguais a 3

Uma árvore binária é uma árvore na qual cada vértice tem no máximo dois filhos, que são chamados filho esquerdo e filho direito. Cada vértice da árvore binária é representado por um nó, e as arestas que conectam os nós expressam as relações de pai-filho.

A árvore binária é uma estrutura de dados muito utilizada em algoritmos e em diversas áreas da ciência da computação, como em algoritmos de busca, estruturas de dados e em inteligência artificial. As árvores binárias são usadas em algoritmos de busca binária, que consistem em encontrar um elemento em uma estrutura de dados ordenada, como um vetor ou uma lista.

Uma árvore binária pode ser percorrida em três ordens diferentes: simétrica (ou in-order), pré-ordem (pre-order) e pós-ordem (post-order). Na ordem simétrica, primeiro é visitado o filho esquerdo, depois o próprio nó e, em seguida, o filho direito. Na pré-ordem, primeiro é visitado o nó, depois o filho esquerdo e, por fim, o filho direito. Na pós-ordem, primeiro são visitados os filhos esquerdo e direito e, por fim, o próprio nó.

As árvores binárias são utilizadas em estruturas de dados, como as árvores de busca binária e as árvores AVL. Além disso, elas são usadas em redes de computadores para representar hierarquias de servidores, e em inteligência artificial para indicar estruturas de decisão em sistemas de aprendizado de máquina.

Observe a seguir alguns pontos importantes:

- as árvores denominadas binárias apresentam, para cada nó, no máximo dois filhos;
- uma árvore binária cheia é uma árvore que tem todos os nós internos com dois filhos e onde todas as folhas estão na mesma profundidade;
- uma árvore binária completa é uma árvore binária que é quase cheia; o nível mais baixo da árvore vai se completando da esquerda para a direita, mas pode ter folhas faltando.

A figura a seguir ilustra uma árvore binária cheia e uma completa, respectivamente.

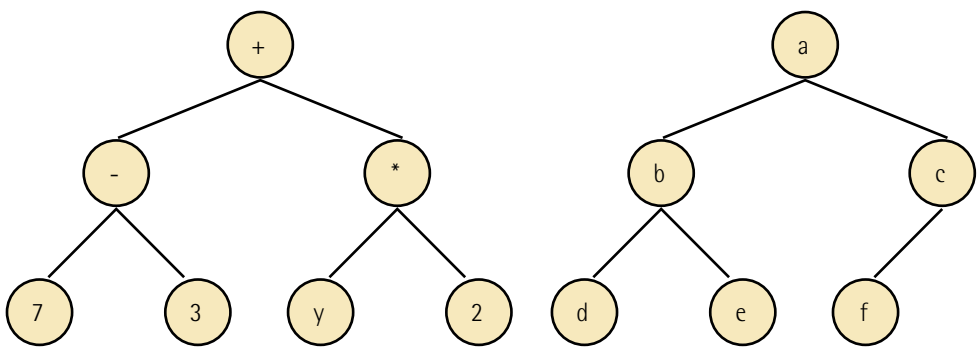
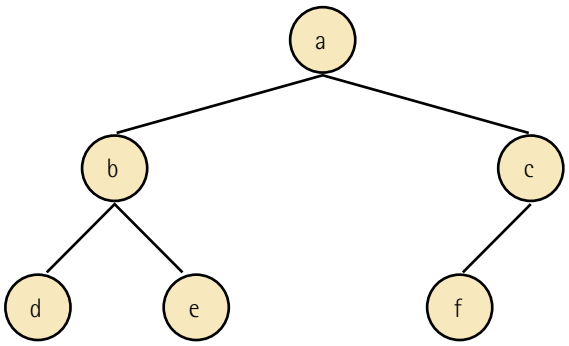


Figura 30

A representação computacional de árvores binárias pode ser efetuada mediante o uso de tabelas.

Exemplo 20

A figura a seguir mostra as representações gráfica e tabular de uma árvore binária. A expressão com ponteiros é seguidamente aduzida. Observa-se que os zeros na tabela se referem a ponteiros nulos. Dessa forma, temos o nó a com seus filhos nó à esquerda b e nó à direita c. Para o nó b com seus filhos, esquerda d e direita e. Para o nó c com seus filhos, esquerda f e direita nulo. Para os nós d, e e f (nós folha) não temos filhos, assim, consideramos 0 para ponteiros nulos.



	Esq	Dir
a	b	c
b	d	e
c	f	0
d	0	0
e	0	0
f	0	0

Figura 31

Exemplo 21

A figura a seguir ilustra a representação gráfica e a correspondente representação utilizando ponteiro de uma árvore binária. O nó raiz a aponta para os nós b e c. Por sua vez, o nó b aponta para os nós d e e. Quanto ao nó c, podemos observar que ele aponta apenas para o nó f, mas temos uma notação para dizer que não há ponteiros. Por sua vez, os nós d, e e f – que são folhas – utilizam a notação para informar que não há ponteiros e, portanto, não apontam para outros nós.

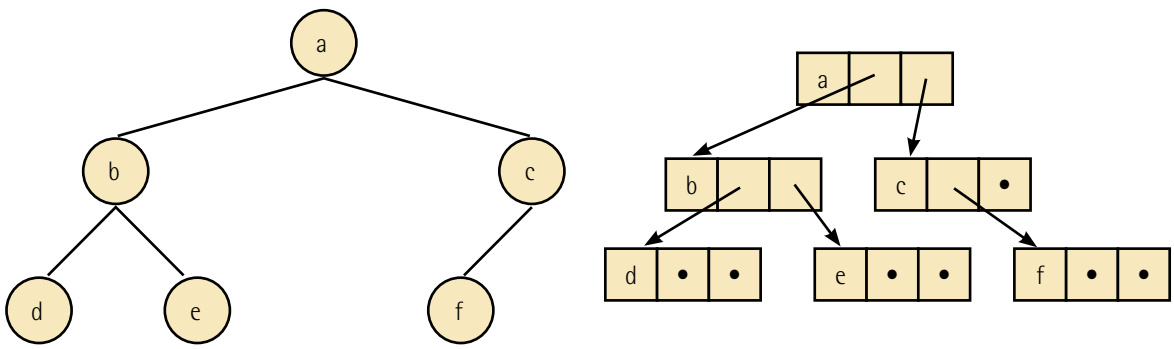


Figura 32

Algoritmos de percurso em uma árvore

Para localizar um elemento em uma árvore, há que se percorrer a árvore, ou seja, visitar todos os seus nós. Os três algoritmos mais comuns são os percursos em pré-ordem, em ordem simétrica e em pós-ordem. A seguir são apresentados esses três algoritmos, segundo Gersting (2014).

No percurso em **pré-ordem**, a raiz é visitada primeiro e depois processam-se as subárvores, da esquerda para a direita.

```
Pré-Ordem(árvore T)

//Escreve os nós de uma árvore com raiz r em pré-ordem

                a,b,d,e,c,f,h,i,g

                Escreva(r)

                Para i = 1 até t faça

                        Pré-Ordem(Ti)

                Fim do Para

Fim Pré-Ordem
```

No percurso em **ordem simétrica**, a subárvore da esquerda é percorrida em ordem simétrica, depois a raiz é visitada e, seguidamente, as outras subárvores são visitadas da esquerda para a direita, sempre em ordem simétrica.

```
Ordem_Simétrica(árvore T)

//Escreve os nós de uma árvore com raiz r em simétrica

Ordem_Simétrica(Ti)

Escreva(r)

        Para i = 2 até t faça

                Ordem_Simétrica(Ti)

        Fim do Para

Fim Ordem_Simétrica
```

No percurso em **pós-ordem**, a raiz é a última a ser visitada após o percurso, em pós-ordem, de todas as subárvores da esquerda para a direita.

```
Pós-Ordem(árvore T)

//Escreve os nós de uma árvore com raiz r em pós-ordem

        Para i = 1 até t faça

                Pós-Ordem(Ti)

        Fim do Para

        Escreva(r).

Fim Pós-Ordem
```

Exemplo 22

A seguir são acentuadas as sequências de nós visitados pela execução dos algoritmos de percurso pré-ordem, ordem simétrica e pós-ordem para a árvore ilustrada na figura 33.

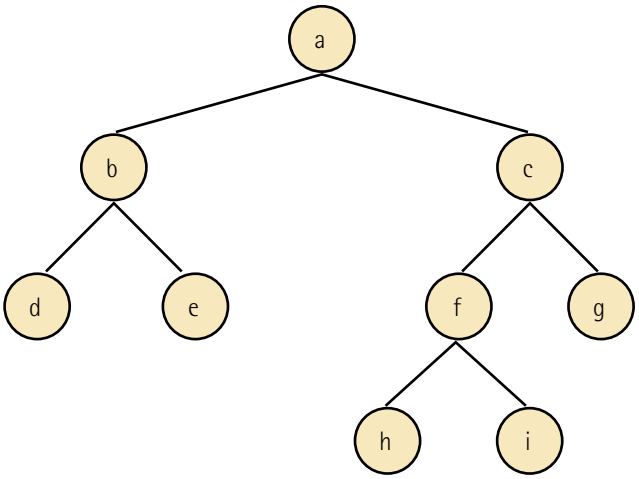


Figura 33 – Árvore

- **percurso em pré-ordem**: a, b, d, e, c, f, h, i, g;
- **percurso em ordem simétrica**: d, b, e, a, h, f, i, c, g;
- **percurso em pós-ordem**: d, e, b, h, i, f, g, c, a.

Exemplo 23

Considere a árvore representada na figura a seguir:

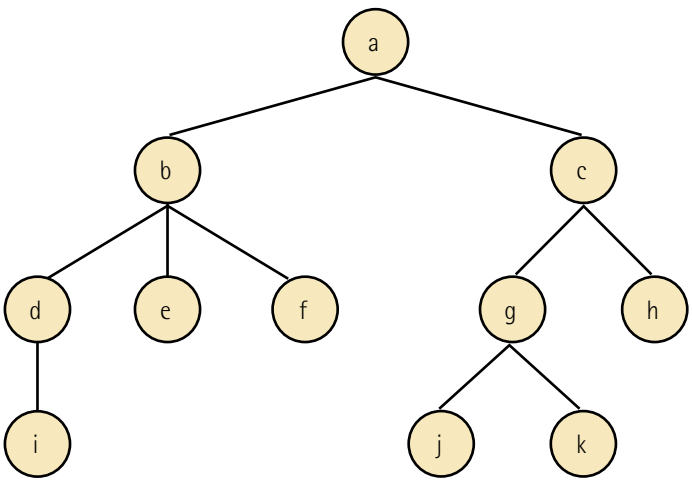


Figura 34

A sequência de nós visitados para cada percurso é:

- **pré-ordem**: a, b, d, i, e, f, c, g, j, k, h;
- **ordem simétrica**: i, d, b, e, f, a, j, g, k, c, h;
- **pós-ordem**: i, d, e, f, b, j, k, g, h, c, a.

Exemplo 24

Considere a árvore indicada na figura a seguir:

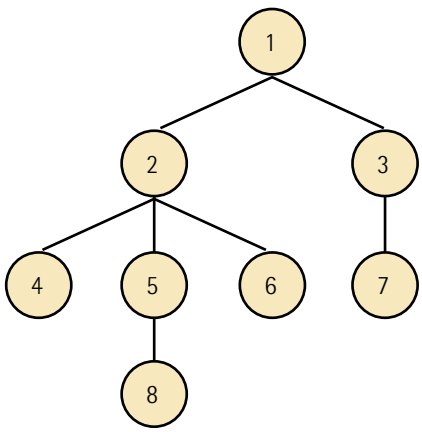


Figura 35

A sequência de nós visitados para cada percurso é:

- **pré-ordem**: 1, 2, 4, 5, 8, 6, 3, 7;
- **ordem simétrica**: 4, 2, 8, 5, 6, 1, 7, 3;
- **pós-ordem**: 4, 8, 5, 6, 2, 7, 3, 1.

Representação de expressões algébricas

A teoria dos grafos pode ser usada para representar expressões algébricas por meio de uma estrutura de dados chamada árvore de expressão. Uma árvore de expressão é uma árvore binária em que cada nó é um operador ou um operando, e os operadores indicam as operações que serão aplicadas aos operandos. Cada operando é uma folha da árvore, enquanto os operadores são os nós internos.

Para criar a árvore de expressão, primeiro precisamos converter a expressão algébrica em uma notação pós-fixa, também conhecida como notação polonesa reversa. Essa notação é conveniente para criar a árvore de expressão, pois permite que os operadores sejam colocados diretamente como nós na árvore, e os operandos sejam adicionados como folhas.

Podemos agora construir a árvore de expressão a partir da notação pós-fixa usando um algoritmo de percurso em profundidade. O algoritmo começa adicionando o primeiro operador da notação pós-fixa como a raiz da árvore; em seguida, adiciona os dois operandos que precedem o operador como filhos da raiz. Esse processo é repetido até que todos os elementos da notação pós-fixa sejam processados.

Depois que a árvore de expressão é criada, podemos realizar operações na expressão, como avaliação e simplificação. Também podemos usar a árvore para criar uma representação visual da expressão, o que pode ser útil para fins de depuração ou para a apresentação de resultados em um formato legível para humanos.

Observando a figura a seguir, tem-se que:

- o percurso pré-ordem +-73*y2 resulta na notação prefixa ou notação polonesa;
- o percurso ordem simétrica 7-3+y*2 resulta na notação infixa;
- o percurso pós-ordem 73-y2*+ resulta na notação pós-fixa ou notação polonesa reversa.

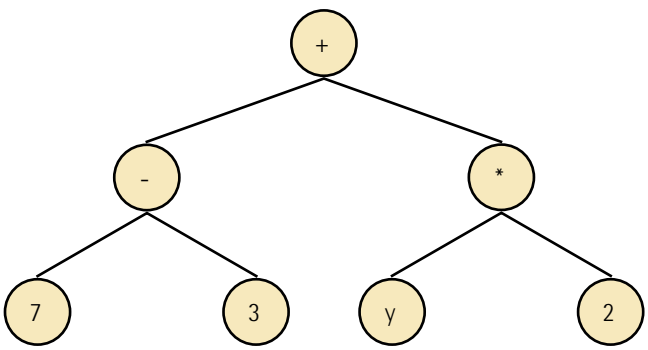


Figura 36

Podemos utilizar a teoria dos grafos para representar expressões algébricas por meio de árvores de expressão, o que permite realizar operações na expressão e criar uma representação visual dela.

Observação

Notação polonesa, também conhecida como notação prefixa ou notação de prefixo polonês, é uma forma de escrever expressões matemáticas ou lógicas sem o uso de parênteses para indicar a ordem de operações. Foi inventada pelo matemático polonês Jan Łukasiewicz na década de 1920.

Na notação polonesa, os operadores são colocados antes dos operandos em vez de entre eles, como é comum na notação infixada utilizada na matemática convencional. Por exemplo, na notação

infixada, a expressão 3 + 4 é escrita como 3 + 4, enquanto na notação polonesa a mesma expressão é escrita como + 3 4.

Além disso, na notação polonesa, a ordem de avaliação das operações é determinada pela ordem em que os operadores aparecem na expressão. Por exemplo, na expressão + * 3 4 5, primeiro é feita a multiplicação de 3 e 4, e depois a adição de 12 e 5 é realizada, resultando em 17.

A notação polonesa pode ser útil para a implementação de linguagens de programação e sistemas de computador, pois elimina a necessidade de usar parênteses para indicar a ordem de operações, o que pode reduzir erros de interpretação e tornar as expressões mais fáceis de serem avaliadas por computadores. Além disso, a notação polonesa pode ser usada em algumas calculadoras científicas e matemáticas, sendo frequentemente aplicada em problemas de lógica e teoria dos conjuntos.

3 CAMINHO DE EULER E CAMINHO HAMILTONIANO

O caminho de Euler visita todas as arestas de um grafo exatamente uma vez. Em outras palavras, passa por todos os vértices do grafo seguindo as arestas sem repetição. Ele é chamado de caminho de Euler em homenagem ao matemático suíço Leonhard Euler, que estudou problemas de caminhos em grafos no século XVIII.

O caminho hamiltoniano, por sua vez, visita todos os vértices de um grafo exatamente uma vez. Ou seja, passa por todos os vértices do grafo sem repetição. Ele é chamado de caminho hamiltoniano em homenagem ao matemático irlandês William Rowan Hamilton, que estudou problemas de ciclos em grafos no século XIX.

3.1 Caminho de Euler

Um caminho de Euler, em um grafo G, usa cada arco em G exatamente uma vez.

No **teorema sobre os nós ímpares em um grafo**, o número de nós ímpares em qualquer grafo é par.

Já no **teorema sobre os caminhos de Euler**, existe um caminho de Euler em um grafo conexo se, e somente se, não existirem nós ímpares ou existirem, exatamente, dois nós ímpares. Quando não houver nós ímpares, o caminho pode começar em qualquer nó; no caso de dois nós ímpares, o caminho precisa começar em um deles e terminar no outro.

Exemplo 25

Para o grafo G, o número de nós ímpares é 4, então não há o caminho de Euler.

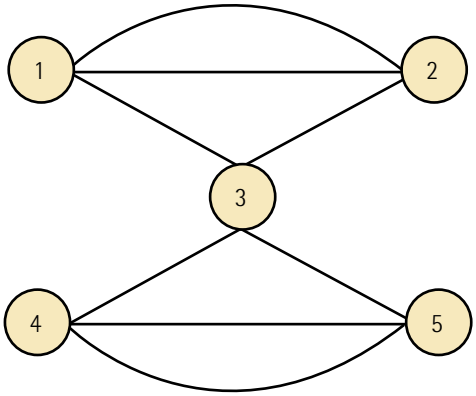


Figura 37

Adaptada de: Gersting (2014).

Agora considere o grafo da figura a seguir:

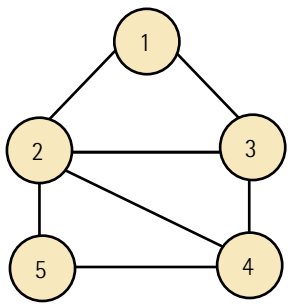


Figura 38

Tem-se que:

- grau (1) = grau (5) = 2;
- grau (2) = 4;
- grau (3) = grau (4) = 3.

Existem, portanto, dois nós ímpares, a saber, os nós 3 e 4. Assim, existe um caminho de Euler, que deve ter seus extremos entre os dois nós: 3-1-2-5-4-2-3-4.

A figura a seguir ilustra a matriz de adjacência do grafo, a partir da qual calcula-se o vetor grau. Ao se inspecionar o vetor grau, é possível contar o número de nós ímpares e, portanto, identificar se se trata de um grafo com o caminho de Euler.

$$\blacksquare\ M=\begin{bmatrix}0&1&1&0&0\\1&0&1&1&1\\1&1&0&1&0\\0&1&1&0&1\\0&1&0&1&0\end{bmatrix}\Rightarrow \text{grau}=\begin{bmatrix}2\\4\\3\\3\\2\end{bmatrix}.$$

Figura 39

Segundo Gersting (2014), encontra-se o algoritmo que determina se existe um caminho de Euler em um grafo conexo com matriz de adjacência A, conforme destacado a seguir.

```
CaminhoDeEuler (matriz n x n A):
// Determina se existe um Caminho de Euler em um grafo conexo com a matriz;
// de adjacência A.
Variáveis locais:
Inteiro total;                  // número de nós ímpares encontrados, até, agora;
Inteiro grau;                   // o grau de um nó;
Inteiros: i, j;                 // índice dos arranjos.
        Total = 0.
        Enquanto total < = 2 e i < 0, faça:
                Grau = 0
                Para j = 1, até n, faça:
                        Grau = grau + A[i, j] // encontra o grau nó i.
                Fim do para:
                Se ímpar (grau), então:
                        Total = Total + 1  // outro nó de grau ímpar // encontrado.
                Fim do se.
                I = i + 1
        Fim do enquanto:
        Se total > 2, então:
                Escreva ("Não existe um caminho de Euler").
        Se não:
                Escreva ("Existe um caminho de Euler").
        Fim do se.
Fim do CaminhoDeEuler.
```

3.2 Caminho hamiltoniano

O problema do circuito hamiltoniano corresponde a verificar se existe um circuito que percorre todos os nós de um grafo uma única vez.

Ao contrário do caminho de Euler, não existe um critério simples para determinar a existência de um circuito hamiltoniano. Trata-se de um problema cuja solução conhecida é de ordem combinatória, ou seja, em uma situação de pior caso, para um grafo com n nós, deverão ser considerados n! permutações dos nós.

Um caminho hamiltoniano passa por todos os vértices do grafo sem repeti-los. É importante destacar que nem todo grafo possui um caminho hamiltoniano. Um exemplo de grafo que não possui um caminho hamiltoniano é um grafo bipartido completo com um número ímpar de vértices.

Encontrar um caminho hamiltoniano em um grafo é um problema NP-completo, o que significa que não há um algoritmo conhecido que possa resolver o problema de forma eficiente para todos os casos. No entanto, existem alguns algoritmos heurísticos que podem encontrar soluções aproximadas para o problema.

Observação

Algoritmos heurísticos são técnicas de resolução de problemas que utilizam métodos aproximados para encontrar soluções aceitáveis em um tempo razoável. Ao contrário de algoritmos exatos, que garantem encontrar a solução ótima de um problema, os heurísticos podem não garantir que a solução encontrada seja ótima, mas são capazes de encontrar soluções aproximadas que podem ser úteis na prática.

Um exemplo de algoritmo para encontrar um caminho hamiltoniano em um grafo é o algoritmo do vizinho mais próximo, que começa em um vértice inicial e, em cada passo, escolhe o vértice mais próximo que ainda não foi visitado. Outro algoritmo popular é o algoritmo 2-opt, que começa com um caminho qualquer e tenta melhorar a solução removendo duas arestas e adicionando outras duas de uma maneira que melhore a solução.

A seguir, destaca-se um pseudocódigo de um algoritmo heurístico para encontrar um caminho hamiltoniano em um grafo utilizando o algoritmo do vizinho mais próximo:

1. Escolha um vértice inicial v.

2. Adicione v ao caminho hamiltoniano.

3. Marque v como visitado.

4. Enquanto houver vértices não visitados no grafo:

    a. Encontre o vértice não visitado mais próximo de v.

    b. Adicione esse vértice ao caminho hamiltoniano.

    c. Marque o vértice como visitado.

    d. Atualize v para ser o último vértice adicionado ao caminho hamiltoniano.

5. Se o último vértice adicionado ao caminho hamiltoniano estiver conectado ao vértice inicial v, retorne o caminho hamiltoniano encontrado. Caso contrário, retorne que não há caminho hamiltoniano no grafo.

Esse algoritmo começa escolhendo um vértice inicial qualquer e adicionando-o ao caminho hamiltoniano. Em seguida, o algoritmo percorre o grafo escolhendo sempre o vértice mais próximo do último vértice adicionado, adicionando-o também ao caminho. O algoritmo continua adicionando vértices até que todos sejam visitados ou não seja possível encontrar um vértice não visitado conectado ao último adicionado.

É importante notar que esse algoritmo é uma heurística e pode não encontrar um caminho hamiltoniano para todos os casos. Além disso, a ordem em que os vértices são visitados pode afetar o resultado do algoritmo, ou seja, o caminho hamiltoniano encontrado pode depender do vértice inicial escolhido.

4 ALGORITMOS DE PERCURSO

O percurso em dado grafo G simples e conexo consiste em escrever todos os nós em certa ordem.

A seguir serão apresentados algoritmos de percurso em nível e em profundidade. Também serão acentuadas duas aplicações para o percurso destinado à busca em profundidade.

4.1 Percurso em nível

A busca em nível começa em um nó arbitrário, anota todos os seus nós adjacentes e seguidamente os nós adjacentes àqueles anteriormente encontrados, e assim por diante.

Segundo Gersting (2014), o algoritmo de percurso em nível em um grafo é aduzido conforme indicado a seguir.

```
EmNível (grafo G; nó a):
// Escreve os nós do grafo G em ordem de nível a partir do nó a.
Variáveis locais: Fila de nós F:
Inicialize F como sendo vazio;
                Marque a como tendo sido visitado;
                Escreva (a);
                Inserir (a, F).
                Enquanto F não é vazio, faça:
                        Para cada nó n adjacente à frente (F), faça:
                                Se n não foi visitado, então:
                                        Marque n como tendo sido visitado;
                                        Escreva n;
                                        Inserir (n, F).
                                Fim_se.
                        Fim_para.
                Retirar (F).
                Fim_enquanto.
Fim EmNivel.
```

Agora considere o seguinte grafo:

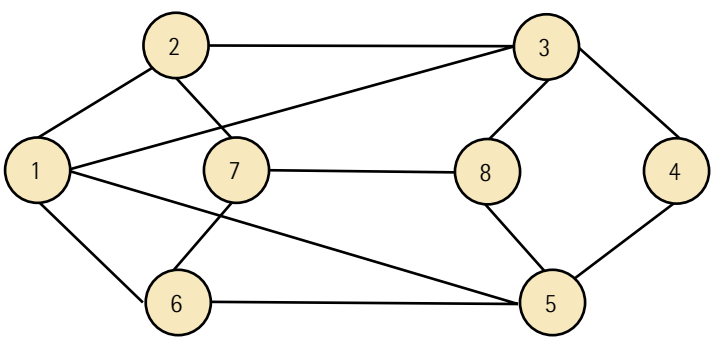


Figura 40

O percurso em nível a partir do nó 1 resulta na seguinte lista de nós: 1, 2, 3, 6, 5, 7, 4, 8.

4.2 Busca em largura

A busca em largura (BFS – Breadth-First Search) é um algoritmo para percorrer um grafo de forma sistemática e explorar todos os seus vértices e arestas de maneira ordenada. A ideia básica é visitar todos os vértices de um mesmo nível antes de avançar para os vértices de níveis mais profundos. A seguir, destaca-se uma descrição geral do algoritmo de busca em largura.

- Comece a busca em um vértice inicial do grafo, marcando-o como visitado.
- Adicione o vértice inicial em uma fila.
- Enquanto a fila não estiver vazia:
  - Remova um vértice da fila.
  - Visite todos os vizinhos desse vértice que ainda não foram visitados e marque-os como visitados.
  - Adicione todos os vizinhos visitados na fila.
- Quando não houver mais vértices na fila, a busca em largura estará concluída.

O processo acentuado é repetido até que todos os vértices do grafo sejam visitados. A fila é usada para manter o controle dos vértices que ainda precisam ser visitados. Quando um vértice é visitado, todos os seus vizinhos que ainda não foram visitados são adicionados à fila para que eles possam ser visitados posteriormente.

A busca em largura pode ser utilizada para diversas finalidades, como encontrar caminhos mais curtos em um grafo não ponderado ou verificar a conectividade de um grafo, entre outras aplicações. Esse algoritmo possui uma complexidade de tempo de O(V+E), onde V é o número de vértices e E é o número de arestas do grafo.

Na prática, a busca em largura é geralmente implementada utilizando uma estrutura de dados fila, que armazena os vértices que ainda não foram visitados em ordem de descoberta. Quando um vértice é descoberto, ele é adicionado à fila e, quando é visitado, é removido da fila, a qual garante que os vértices são visitados em ordem de proximidade ao vértice inicial.

Um aspecto importante da busca em largura é a ordem em que os vértices são visitados. Na maioria das implementações, os vértices são visitados em ordem crescente de distância em relação ao vértice inicial. Isso significa que os vértices que estão mais próximos do vértice inicial são visitados primeiro. No entanto, a ordem de visita pode ser personalizada de acordo com a necessidade específica do problema que está sendo resolvido.

Uma limitação da busca em largura é que ela não é eficiente para encontrar caminhos mais curtos em grafos ponderados, pois não leva em conta o peso das arestas. Para isso, deve-se utilizar

algoritmos como o de Dijkstra ou o de Bellman-Ford. Além disso, a busca em largura pode levar a soluções subótimas em alguns casos, pois nem sempre encontra o caminho mais curto.

4.3 Busca em profundidade

A busca em profundidade (DFS – Depth-First Search) é um algoritmo para percorrer um grafo de forma sistemática e explorar todos os seus vértices e arestas de maneira ordenada. A ideia básica é visitar todos os vértices de um ramo do grafo antes de voltar para a raiz do ramo e explorar outro ramo. A seguir, destaca-se uma descrição geral do algoritmo de busca em profundidade:

1. Comece a busca em um vértice inicial do grafo, marcando-o como visitado.
2. Para cada vizinho do vértice inicial que ainda não foi visitado, repita os passos 3 a 5.
3. Visite o vizinho e marque-o como visitado.
4. Recursivamente, visite todos os vizinhos desse vértice que ainda não foram visitados e marque-os como visitados.
5. Retorne ao vértice atual.
6. Quando não houver mais vértices a serem visitados, a busca em profundidade estará concluída.

Observação

Recursividade é um conceito em programação e matemática que envolve a definição de uma função que se chama repetidamente, com base em uma condição de parada, até que o resultado seja alcançado.

Em outras palavras, uma função recursiva chama a si mesma dentro de sua própria definição. Quando a função é chamada, ela executa o código dentro dela, o que pode incluir uma chamada a si mesma com argumentos diferentes. Esse processo continua até que uma condição de parada seja atingida, momento em que a função começa a retornar valores e desempilhar as chamadas recursivas anteriores.

A recursividade é frequentemente usada para resolver problemas que possuem uma estrutura recursiva, como cálculos em árvores, listas encadeadas ou estruturas matemáticas como fatoriais e sequências de Fibonacci. No entanto, a recursividade pode ser ineficiente e levar a problemas de tempo de execução em casos extremos, pois pode exigir uma grande quantidade de espaço na pilha de chamadas da função.

O processo acentuado é repetido até que todos os vértices do grafo sejam visitados. A busca em profundidade usa uma abordagem recursiva para explorar todos os vértices do grafo. Cada vez que um

vértice é visitado, a busca em profundidade avança para um nível mais profundo do grafo, explorando o ramo mais profundo antes de voltar para o ramo anterior.

A busca em profundidade pode ser aplicada para diversas finalidades, como encontrar ciclos ou verificar a conectividade de um grafo, entre outras aplicações. Esse algoritmo possui uma complexidade de tempo de O(V+E), onde V é o número de vértices e E é o número de arestas do grafo.

Na prática, a busca em profundidade é geralmente implementada utilizando uma pilha, que armazena os vértices que ainda não foram visitados. Quando um vértice é visitado, ele é removido da pilha e seus vizinhos são adicionados a ela. A pilha garante que os vértices são visitados em ordem de profundidade.

Um aspecto importante da busca em profundidade é a ordem em que os vértices são visitados. Na maioria das implementações, eles são visitados em ordem crescente de profundidade em relação ao vértice inicial. Isso significa que os vértices mais profundos do grafo são visitados primeiro. No entanto, a ordem de visita pode ser personalizada de acordo com a necessidade específica do problema que está sendo resolvido.

Uma limitação da busca em profundidade é que ela pode levar a soluções subótimas em alguns casos, pois nem sempre encontra o caminho mais curto. Além disso, a busca em profundidade pode entrar em ciclos infinitos em grafos com ciclos se não for implementada corretamente.

Para evitar a ocorrência de ciclos infinitos, é necessário utilizar alguma forma de controle de visitação dos vértices. Isso pode ser feito através da marcação dos vértices como visitados durante a execução da busca em profundidade. Quando um vértice já foi visitado, ele não será mais adicionado à pilha, evitando que o algoritmo entre em um ciclo infinito.

Outra limitação da busca em profundidade é que ela pode ser menos eficiente do que a busca em largura para encontrar caminhos mais curtos em grafos não direcionados, pois a ordem de visita dos vértices não leva em conta a distância em relação ao vértice inicial. No entanto, para certos tipos de problemas, a busca em profundidade pode ser mais eficiente do que a busca em largura.

Em resumo, a busca em profundidade é um algoritmo de percurso de grafos que utiliza uma abordagem recursiva para explorar todos os vértices e arestas do grafo. É importante lembrar que a escolha entre a busca em largura e a busca em profundidade depende do problema que está sendo resolvido e das características do grafo em questão.

4.4 Ordenação topológica

É um conceito importante em teoria dos grafos que se aplica a grafos direcionados acíclicos (DAGs–Directed Acyclic Graphs). Uma ordenação topológica é uma ordenação linear dos vértices do grafo que respeita a direção das arestas. Em outras palavras, se existe uma aresta direcionada do vértice u para o vértice v, então u aparece antes de v na ordenação.

A ordenação topológica é usada em problemas em que é necessário determinar uma ordem de execução que respeite as dependências entre tarefas. Por exemplo, imagine que você precisa planejar as atividades de um projeto, em que algumas atividades só podem ser executadas após a conclusão de outras atividades. Nesse caso, podemos representar as atividades como vértices de um DAG, e as dependências entre as atividades como arestas direcionadas. Uma ordenação topológica do DAG indica uma ordem de execução das atividades que respeita as dependências entre elas.

É importante ressaltar que nem todos os grafos direcionados são DAGs, pois alguns possuem ciclos. Em um grafo com ciclo, não é possível definir uma ordenação topológica, pois não é possível respeitar a direção das arestas. Por esse motivo, a ordenação topológica só se aplica a DAGs.

Existem diferentes algoritmos para encontrar uma ordenação topológica em um DAG. Um dos algoritmos mais simples é baseado em uma busca em profundidade. O algoritmo funciona da seguinte maneira:

1. Inicialmente, marque todos os vértices do grafo como não visitados.
2. Escolha um vértice não visitado e execute uma busca em profundidade a partir dele.
3. À medida que os vértices forem visitados durante a busca em profundidade, adicione-os a uma lista.
4. Retorne à etapa 2 e escolha outro vértice não visitado. Repita até que todos os vértices tenham sido visitados.

Ao final do algoritmo, a lista criada contém uma ordenação topológica do grafo. Esse algoritmo possui uma complexidade de tempo de O(V+E), onde V é o número de vértices e E é o número de arestas do grafo.

A ordenação topológica tem diversas aplicações, como em programação dinâmica, planejamento de atividades, organização de tarefas em fluxos de trabalho, entre outras.

Uma ordenação topológica de um DAG G = (V, E) é uma ordenação linear de todos os seus vértices, tal que se G contém uma aresta (u, v), então u aparece antes de v na ordenação (CORMEN *et al.*, 2012). Naturalmente, se o grafo contém um ciclo, nenhuma ordenação topológica é possível.

Muitas aplicações usam DAGs para indicar precedências entre eventos.

O algoritmo simples a seguir, apresentado por Cormen *et al.* (2012), ordena topologicamente um DAG:

```
TOPOLOGICAL-SORT(G)

        chamar DFS(G) para calcular o tempo de término v.f para cada vértice v

        à medida que cada vértice é terminado, inserir o vértice à frente de uma
lista ligada

return a lista ligada de vértices
```

4.5 Componentes fortemente conexas

Um grafo dirigido é fortemente conexo se cada vértice pode ser alcançado de qualquer outro vértice. As componentes fortemente conexas de um grafo dirigido são as classes de equivalência de vértices sob a relação "são mutuamente acessíveis". Um grafo dirigido é fortemente conexo se tem somente uma componente fortemente conexa. Por exemplo, o grafo da figura 41A tem três componentes fortemente conexos: {1, 2, 4, 5}, {3} e {6}. Todos os pares de vértices em {1, 2, 4, 5} são mutuamente acessíveis. Os vértices {3, 6} não formam uma componente fortemente conexa, visto que o vértice 6 não pode ser alcançado do vértice 3.

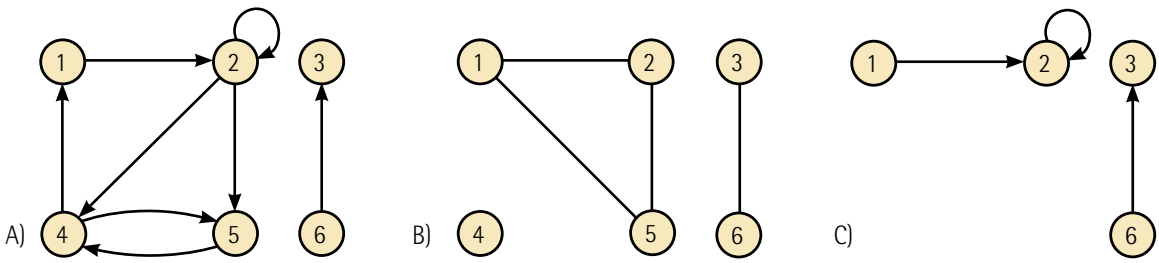


Figura 41

Adaptada de: Cormen *et al.* (2012).

Resumo

Vimos nesta unidade diversas considerações sobre grafos. Um grafo é uma estrutura matemática composta de um conjunto de vértices (ou nós) e um conjunto de arestas (ou arcos) que conectam esses vértices. Os vértices representam objetos ou entidades, enquanto as arestas expressam as relações entre esses objetos ou entidades. Os grafos são utilizados em diversas áreas, como ciência da computação, matemática, física, biologia e engenharia, para modelar sistemas complexos e suas interações. Dependendo das características das arestas e dos vértices, os grafos podem ser direcionados ou não direcionados, ponderados ou não ponderados, entre outros tipos.

Aprendemos que uma árvore é uma estrutura matemática formada por um conjunto de vértices (ou nós) conectados por arestas (ou ramificações) que satisfazem certas propriedades. Em uma árvore, um dos vértices é designado como raiz, e cada vértice tem no máximo um pai e vários filhos. As arestas representam a conexão hierárquica entre os vértices e são sempre direcionadas da raiz para as folhas. As árvores são usadas em diversas áreas, como ciência da computação, matemática, biologia e engenharia, para modelar hierarquias e estruturas de dados organizadas.

Depois, apresentamos os caminhos de um grafo. O caminho de Euler percorre cada aresta exatamente uma vez. Em outras palavras, passa por todos os vértices do grafo. Já um caminho hamiltoniano percorre cada vértice exatamente uma vez. Em outras palavras, passa por todos os vértices do grafo. Ambos os conceitos são importantes em teoria dos grafos e têm aplicações práticas em diversas áreas, como roteamento de redes, programação de computadores e otimização de processos. É importante destacar que nem todo grafo possui um caminho hamiltoniano ou um caminho de Euler, e que encontrar esses caminhos em grafos de grande porte pode ser um problema computacionalmente difícil.

Também acentuamos os algoritmos de percurso. Eles são utilizados em grafos para visitar todos os vértices e/ou arestas do grafo em uma determinada ordem. Existem dois tipos principais de algoritmos de percurso: busca em largura (BFS) e busca em profundidade (DFS). Na busca em largura, o algoritmo visita todos os vértices que estão a uma certa distância de um vértice inicial primeiro, isto é, antes de se mover para os vértices que estão a uma distância maior. Essa estratégia garante que o algoritmo visite todos os vértices em ordem crescente de distância em relação ao vértice inicial. Na busca em profundidade, o algoritmo visita o

vértice inicial e, em seguida, explora o máximo possível do grafo em uma direção antes de fazer um retrocesso e explorar outras direções. Isso garante que o algoritmo explore o grafo o mais profundamente possível antes de voltar para explorar outras áreas. Os algoritmos de percurso são usados em diversas aplicações, como busca em grafos, ordenação topológica e solução de problemas de conectividade em redes.

Exercícios

Questão 1. (Cespe-Cebraspe 2018, adaptada) Considere o grafo disposto a seguir.

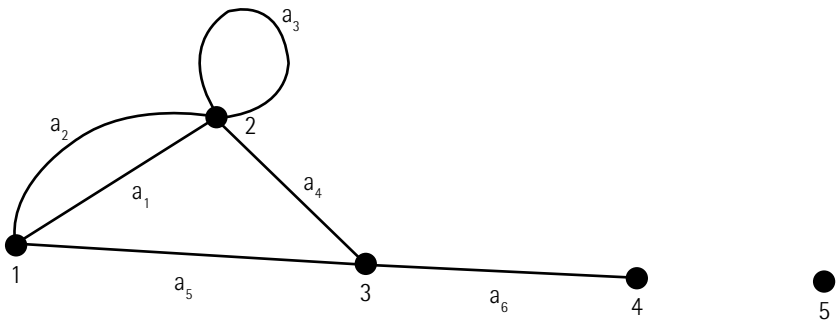


Figura 42

A respeito do grafo apresentado, avalie as afirmativas.

I – Os nós 1 e 4 são adjacentes.

II – O nó 5 é adjacente a si mesmo.

III – Os nós 2 e 3 têm grau 3.

IV – O grafo não pode ser classificado como conexo.

É correto o que se afirma em:

A) I, apenas.

B) III, apenas.

C) IV, apenas.

D) I, II e IV, apenas.

E) I, II, III e IV.

Resposta correta: alternativa C.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: dois nós em um grafo são ditos adjacentes se ambos são extremidades de algum arco. No grafo da questão, os nós 1 e 4 não são conectados diretamente por apenas um arco, por mais que existam caminhos entre eles. Desse modo, eles não são considerados adjacentes.

II – Afirmativa incorreta.

Justificativa: um nó que é terminal de um laço é adjacente a si próprio. No nó 5 não há qualquer laço, já que ele não é conectado a ele mesmo por uma aresta. Desse modo, não é possível dizer que o nó 5 é adjacente a si mesmo.

III – Afirmativa incorreta.

Justificativa: o grau de um nó é o número de arcos que terminam naquele nó. O nó 3 tem grau 3, já que três arcos chegam a ele. O nó 2 tem grau 4, já que quatro arcos chegam a ele.

IV – Afirmativa correta.

Justificativa: um grafo é classificado como conexo se existe um caminho de qualquer nó para qualquer outro. Em outras palavras, em um grafo conexo, existe pelo menos um caminho entre cada par de nós do grafo. No grafo da questão, o nó 5 não está conectado a nenhum outro. Isso faz com que o grafo seja classificado como desconexo.

Questão 2. Avalie a figura a seguir, que representa um grafo.

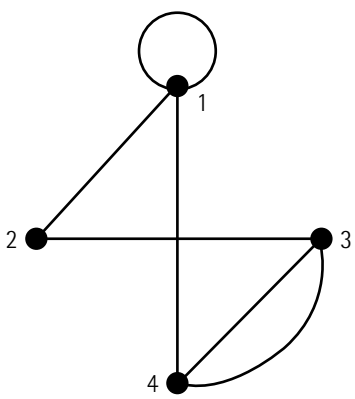
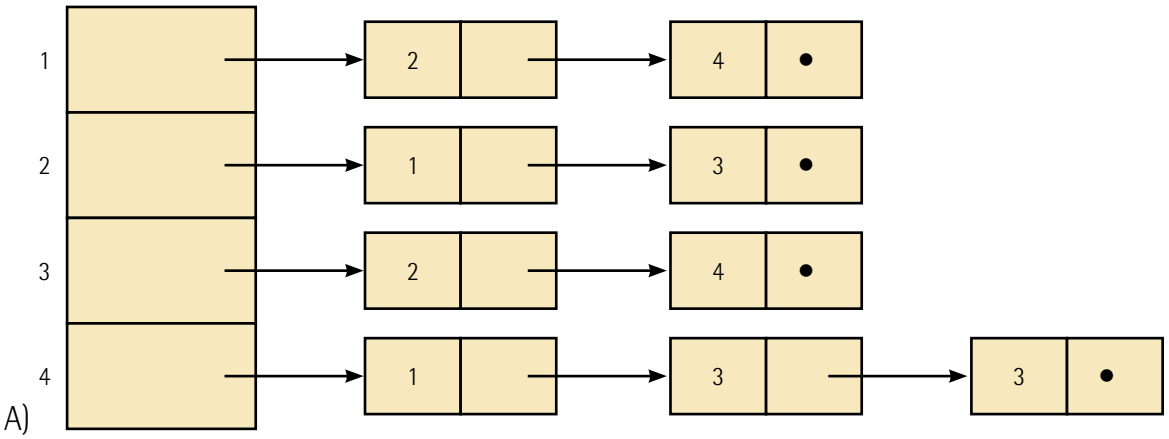


Figura 43

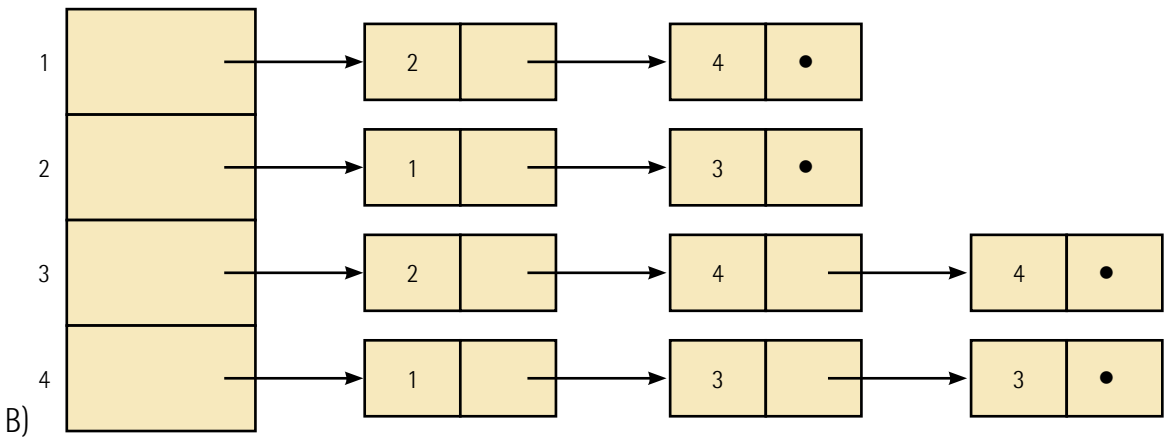
GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Rio de Janeiro: LTC, 2017. p. 446.

Assinale a alternativa que representa corretamente a lista de adjacência para o grafo em questão.

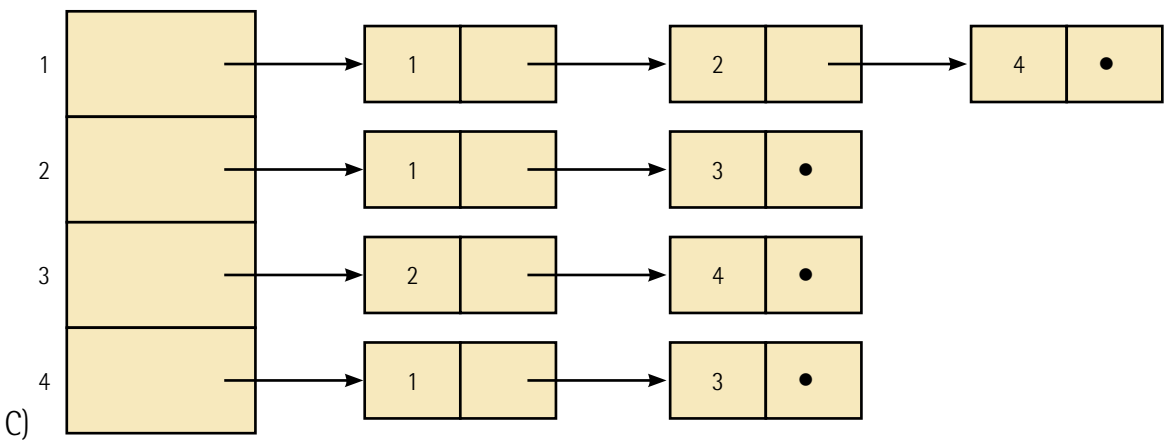
A)



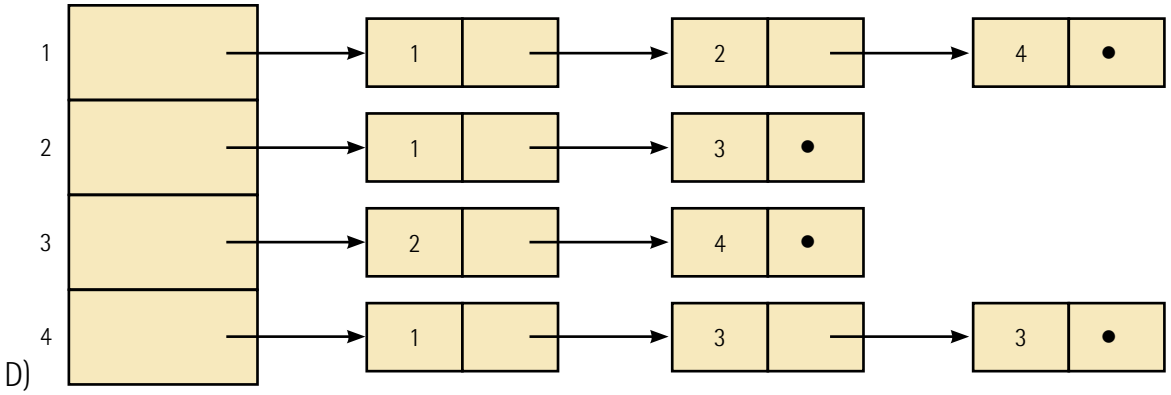
B)



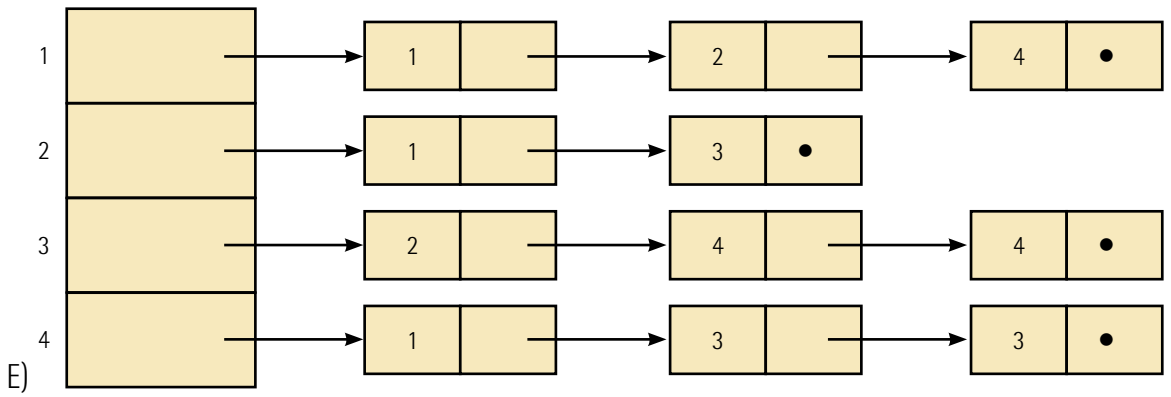
C)



D)



E)



Resposta correta: alternativa E.

Análise da questão

A lista de adjacência para o grafo da questão deve conter uma tabela de ponteiros com quatro elementos, um para cada nó, cada um disposto em uma linha. O ponteiro de cada nó aponta para um nó adjacente, que aponta para outro nó adjacente, e assim por diante. A estrutura de lista de adjacência do grafo do enunciado é mostrada a seguir.

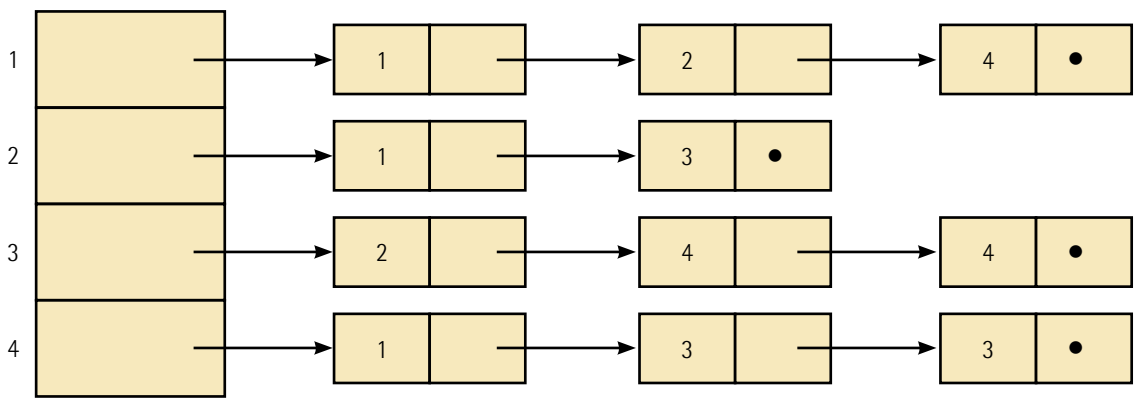


Figura 44

GERSTING, J. L. *Fundamentos matemáticos para a ciência da computação*. Rio de Janeiro: LTC, 2017. p. 448.

Perceba que, para o nó 1, cujo ponteiro é disposto na primeira linha, o ponteiro aponta para o próprio nó 1 (representando o laço do grafo), para o nó 2 (já que há uma aresta ligando o nó 1 ao nó 2) e para o nó 4 (já que há uma aresta ligando o nó 1 ao nó 4).

Para o nó 2, cujo ponteiro é disposto na segunda linha, o ponteiro aponta para o nó 1 (já que há uma aresta ligando o nó 2 ao nó 1) e para o nó 3 (já que há uma aresta ligando o nó 2 ao nó 3).

Para o nó 3, cujo ponteiro é disposto na terceira linha, o ponteiro aponta para o nó 2 (já que há uma aresta ligando o nó 3 ao nó 2) e para o nó 4 duas vezes, já que há duas arestas conectando o nó 3 ao nó 4.

Finalmente, para o nó 4, cujo ponteiro é disposto na quarta linha, o ponteiro aponta para o nó 1 (já que há uma aresta ligando o nó 4 ao nó 1) e para o nó 3 duas vezes, já que há duas arestas conectando o nó 4 ao nó 3.