DATA MINING E GRAPH MINING

SOLUÇÕES EDUCACIONAIS INTEGRADAS

Classificação de grafos

Rafael Leal Martins

OBJETIVOS DE APRENDIZAGEM

- Importar dados de uma base, armazenando-os em formato de grafos.
- Apresentar formas de classificação de grafos a partir de importados de uma base.
- Descrever as técnicas de classificação em grafos.

Introdução

Grafos são uma estrutura de dados muito importante na área de *data mining*, pois são utilizados para representar dados e suas inter-relações. Sua estrutura em rede permite uma visão mais abrangente e abstrata de cenários em que associações de informações são muito frequentes. Eles são, portanto, ferramentas úteis em vários cenários de análise de dados. Na química, por exemplo, pode-se utilizar grafos para representar estruturas moleculares de compostos químicos, com os vértices representando átomos e as arestas representando as ligações entre esses átomos. Esses grafos podem ser analisados para determinar propriedades como toxicidade de uma substância.

Neste capítulo, você vai entender o processo de criação de grafos utilizando a linguagem Python, a partir de dados importados de bancos de dados relacionais e não relacionais (NoSQL). Além disso, vamos tratar dos critérios de classificação de grafos e de formas de aplicar essas técnicas na prática.

Como construir grafos em Python?

Grafos são estruturas de dados compostas de um conjunto de vértices, que representam elementos de dados, e um conjunto de arestas, que representam as associações entre esses vértices. Apesar de grafos serem matematicamente definidos pelos elementos desses dois conjuntos, é muito comum um grafo ter uma representação gráfica, ilustrando melhor a integração dos vértices. A Figura 1 mostra a relação entre essas duas formas de representação e apresentação de um grafo.

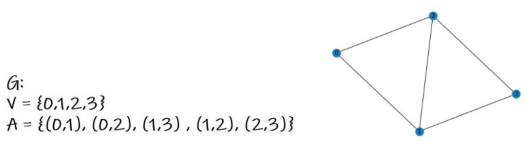

Figura 1. Duas formas de se representar um grafo.

Para você poder usar grafos em seu cotidiano pessoal ou profissional, você terá que implementar essa estrutura por meio de uma linguagem de programação. Aqui, utilizaremos a linguagem Python e um módulo chamado de NetworkX para nos ajudar a criar esses grafos de forma mais simples, eficaz e eficiente.

Módulo NetworkX para construção de grafos

A Python possui um módulo chamado de NetworkX (HAGBERG; SCHULT; SWART, 2008), que tem várias funções para criação, manipulação e até desenho de grafos. O módulo é gratuito e está disponível no *site* oficial, mas também pode ser instalado, de forma mais rápida e direta, pelo utilitário "pip" ou pela plataforma Anaconda. O *site* oficial, porém, possui a documentação completa do módulo (HAGBERG; SCHULT; SWART, 2008). Aqui, utilizaremos apenas algumas de suas funcionalidades, mas você precisa instalar o módulo para o utilizar.

Agora veremos como escrever um *script* Python utilizando o módulo `networkx` para criar o grafo G, mostrado na Figura 2. Note que esse grafo possui três vértices, A, B e C, e três arestas interconectando cada vértice aos outros dois: (A, B), (B, C) e (C, A).

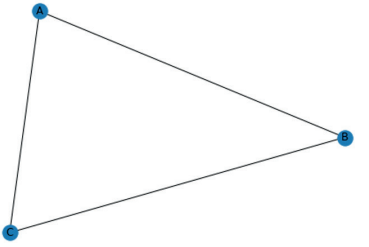

Figura 2. Grafo G criado com o módulo `networkx`.

Fique atento

O grafo G da Figura 2 é um grafo não direcionado, ou seja, não há o conceito de direção na ligação dos vértices. Nesse caso, tanto faz descrever uma aresta como (A,B) ou (B,A). No caso de grafos direcionados, essa direção passa a ser relevante, ou seja: (A,B) é diferente de (B,A).

O *script* é este:

```
1. import networkx as nx

2. G = nx.Graph()

3. G.add_node('A')

4. G.add_node('B')

5. G.add_node('C')

6. G.add_edge('A', 'B')

7. G.add_edge('B', 'C')

8. G.add_edge('C', 'A')

9. nx.draw(G, with_labels=True)
```

Vamos analisar cada linha desse *script*. A linha 1 é necessária para importar o módulo `networkx` para ser utilizado no código. Na linha 2, criamos o grafo G, que inicialmente está vazio (sem vértices, que, na terminologia do NetworkX, são chamados de *nodes*). As linhas 3, 4 e 5 usam a função `add_node` para adicionar os vértices (*nodes*) A, B e C. As linhas 6, 7 e 8 criam as arestas (em inglês, *edges*) com chamadas ao método `add_edge`. Finalmente, a linha 9

desenha o grafo por meio da chamada ao comando `nx.draw`. O argumento `with _ labels` é definido como `True` para que o desenho do grafo mostre os rótulos (*labels*) com os nomes dos vértices.

O NetworkX fornece quatro formas de grafos por meio de classes Python:

1. **Graph:** grafo não direcionado, que ignora arestas múltiplas entre vértices (por exemplo, duas arestas entre A e B), mas aceita *loops* (aresta de um nó para ele mesmo).
2. **DiGraph:** grafos direcionados (com operações específicas para esse tipo de grafo).
3. **MultiGraph:** aceita arestas múltiplas (não direcionadas) entre dois vértices.
4. **MultiDiGraph:** versão direcionada do MultiGrap.

Segundo Hagberg, Schult e Swart (2008), todas as classes de grafos permitem qualquer objeto *hashable* como um nó (*node*). Objetos *hashable* incluem *strings*, tuplas, inteiros e muito mais. Atributos arbitrários de arestas (*edges*), como pesos e rótulos, podem ser associados a uma aresta". Também é possível criar grafos usando listas de vértices e listas de arestas como mostrado no código a seguir:

```
1. import networkx as nx

2. vertices=[1,2,3]

3. arestas=[(1,2), (2,3), (3,1)]

4. G = nx.Graph()

5. G.add _ nodes _ from(vertices)

6. G.add _ edges _ from(arestas)

7. nx.draw(G, with _ labels=True)
```

Note que, nesse exemplo, foram criadas duas listas (`vertices` e `arestas`) com os valores desejados. As arestas podem ser fornecidas por meio de uma lista de tuplas contendo as associações entre os vértices. Na verdade, é possível criar o grafo apenas fornecendo a lista de arestas, como mostrado a seguir:

```
1. import networkx as nx

2. arestas=[('Brasilia','Lisboa'), ('Lisboa','Paris')]
```

```
3. G = nx.Graph()

4. G.add _ edges _ from(arestas)

5. nx.draw(G, with _ labels=True, node _ size=2800)
```

Resultando no grafo mostrado na Figura 3.

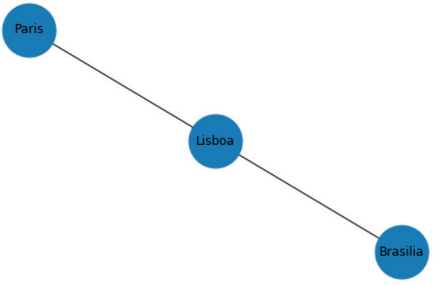

Figura 3. Grafo G criado apenas a partir de uma lista de arestas.

Também podemos definir valores de pesos para as arestas e exibir essas informações, como mostrado no código a seguir:

```
import networkx as nx

G = nx.Graph()

G.add _ node('A')

G.add _ node('B')

G.add _ node('C')

G.add _ edge('A', 'B', weight=1)

G.add _ edge('B', 'C', weight=2)

G.add _ edge('C', 'A', weight=3)

pos = nx.circular _ layout(G)

nx.draw(G, pos, with _ labels=True, node _ size=2800)

labels = nx.get _ edge _ attributes(G,'weight')

print(labels)
```

```
nx.draw_networkx_edge_labels(G, pos=nx.
circular_layout(G))
```

Aqui você pode ver um exemplo do uso de *layouts* para definir a forma visual do grafo a ser gerado. Para definir o peso de cada aresta, foi definido o valor do atributo `weight` para cada uma. Nesse caso, foi adotado o `circular_layout` tanto para o posicionamento dos vértices quanto para o posicionamento do texto contendo os valores dos pesos das arestas. Para exibir os valores dos pesos das arestas, utilizamos `draw_networkx_edge_labels`. O grafo resultante do código acima pode ser visto na Figura 4.

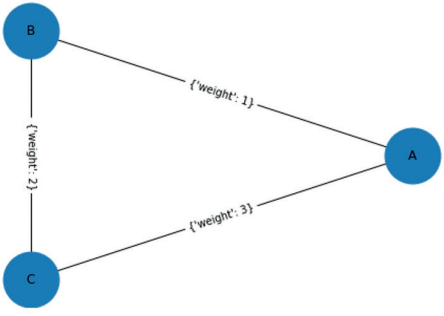

Figura 4. Grafo ponderado (com pesos nas arestas) criado com o módulo `networkx`.

Há inúmeras outras opções para criar e customizar grafos. Infelizmente, não conseguiremos descrever todas aqui. Vale a pena consultar a referência completa da ferramenta, para ver a documentação completa (HAGBERG; SCHULT; SWART, 2008). Agora que você viu o essencial para a criação dos grafos utilizando o módulo NetworkX, vamos aplicar esses conhecimentos no âmbito dos bancos de dados.

Criação de grafos a partir de bancos de dados relacionais

Segundo definição de Chakrabarti (2011), ***graph mining*** é o conjunto de ferramentas e técnicas usadas para analisar as propriedades dos grafos do mundo real, prever como a estrutura e as propriedades de determinado grafo podem afetar alguma aplicação e desenvolver modelos que possam gerar grafos realistas, que correspondam aos padrões encontrados em grafos de interesse do mundo real. Isso significa que os grafos são obtidos a partir de

bancos de dados reais, o que nos leva à seguinte questão: como criar grafos a partir de bancos de dados? Você aprenderá, agora, técnicas para gerar grafos usando Python juntamente ao módulo NetworkX a partir de bancos de dados relacionais e não relacionais.

Vamos começar com um banco de dados relacional, ou seja, um banco em que os dados são armazenados em tabelas (relações) e podem ser consultados por meio da linguagem SQL. Para simplificarmos nossos exemplos, utilizaremos o SQLite: um banco de dados relacional mais simples, que armazena os dados localmente, sem a necessidade de instalar e configurar um servidor de rede. Entretanto, vale lembrar que a ideia por trás da importação independe do banco de dados utilizado.

Nesse exemplo, considere um banco de dados chamado de `ciaAerea.db`, que contém uma tabela chamada de `VOOS`, com dados de distância de voos entre algumas capitais do Brasil, conforme mostrado na Figura 5. A coluna `Weight` contém a distância em km entre as capitais associadas. O nome da coluna no banco não precisa ser necessariamente esse, ele só foi mantido para explicitar que o peso da aresta representa a distância.

	Origem	Destino	Weight
▶	Brasília	Goiânia	209
	Goiânia	Belo Horizonte	906
	Belo Horizonte	São Paulo	586
	São Paulo	Rio de Janeiro	429
	Rio de Janeiro	Belo Horizonte	434
	São Paulo	Brasília	1015

Figura 5. Conteúdo da tabela `VOOS` do banco de dados `ciaAerea.db`.

Você verá agora um exemplo de *script* em Python que lê os registros da tabela `VOOS` e cria, a partir dos dados dessa tabela, um grafo ponderado (com pesos nas arestas).

```
1. import networkx as nx

2. import sqlite3

3. #Conexão com o banco e preparação da tabela com os dados

4. with sqlite3.connect("ciaAerea.db") as con:
```

```
5. cur = con.cursor()

6. cur.execute("CREATE TABLE VOOS(Origem TEXT, Destino TEXT,
Weight REAL)")

7. cur.execute("INSERT INTO VOOS VALUES('Brasília', 'Goiâ-
nia', 209)")

8. cur.execute("INSERT INTO VOOS VALUES('Goiânia', 'Belo
Horizonte', 906)")

9. cur.execute("INSERT INTO VOOS VALUES('Belo Horizonte',
'São Paulo', 586)")

10. cur.execute("INSERT INTO VOOS VALUES('São Paulo', 'Rio
de Janeiro', 429)")

11. cur.execute("INSERT INTO VOOS VALUES('Rio de Janeiro',
'Belo Horizonte',434)")

12. cur.execute("INSERT INTO VOOS VALUES('São Paulo', 'Bra-
sília', 1015)")

13. #Criação do Grafo G a partir da tabela VOOS

14. G = nx.Graph()

15. cur.execute('SELECT * FROM VOOS')

16 G.add _ weighted _ edges _ from(cur)

17. pos = nx.planar _ layout(G)

18. nx.draw(G, pos, with _ labels=True)

19. nx.draw _ networkx _ edge _ labels(G, pos)
```

Basicamente, as linhas de 4 a 13 tratam do estabelecimento da conexão com o banco de dados (`ciaAerea.db`), criam a tabela `VOOS` e inserem alguns registros nessa tabela. Na linha 15, recuperamos os dados da tabela e criamos o grafo na linha 16 a partir da lista retornada pelo objeto cursor com o comando `add _ weighted _ edges _ from`. As linhas 17, 18 e 19 apenas tratam da exibição do grafo, que é mostrado na Figura 6. Vale ressaltar que as posições dos vértices desse grafo não correspondem às posições reais das capitais: trata-se apenas de um diagrama ilustrativo.

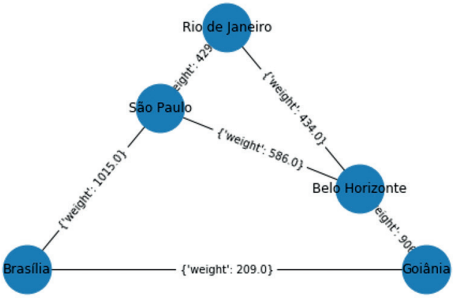

Figura 6. Grafo criado a partir da importação de dados da tabela VOOS.

Criação de grafos a partir de bancos de dados não relacionais (NoSQL)

Você também pode criar grafos a partir de bancos de dados não relacionais. Você pode utilizar o módulo `pymongo` para manipular dados usando o MongoDB. O código a seguir mostra, na linha 1, como se conectar a um servidor MongoDB executando localmente (*localhost*), nas linhas 2 e 3, como acessar, no banco de dados `bdgrafos`, a collection `contatos` e, nas linhas 5 e 6, como inserir, na `collection`, um registro, definido na forma de um dicionário em Python contendo as chaves `nome` e `endereco`.

```
1. import pymongo

2. cliente = pymongo.MongoClient("mongodb://127.0.0.1:27017/")

3. bd = cliente["bdgrafos"]

4. collection = bd["contatos"]

5. contato = {"nome": "Fulano", "endereco": "Rua X"}

6. x = collection.insert _ one(contato)
```

Agora, estenderemos esse exemplo para gerar um grafo direcionado informando quais pessoas residem em quais endereços. A ideia é observar quais são os endereços mais comuns no banco e quais pessoas os utilizam. Para isso, o *script* Python a seguir vai inserir um número maior de registros no MongoDB (linhas 6 a 14), depois recuperar esses registros do banco, por meio do comando `find()`, e criar o grafo direcionado a partir dos dados recuperados, no caso apenas criando arestas com os dados buscados da `collection` (linhas 16 a 18).

```
1. import pymongo
2. import networkx as nx
3. cliente = pymongo.MongoClient("mongodb://127.0.0.1:27017/")
4. bd = cliente["bdgrafos"]
5. collection = bd["contatos"]
6. lista _ contatos = [
7. { "nome": "Fulano", "endereco": "Rua X" },
8. { "nome": "Cicrano", "endereco": "Rua X" },
9. { "nome": "Beltrano", "endereco": "Rua Y" },
10. { "nome": "Alano", "endereco": "Rua Y" },
11. { "nome": "Mengano", "endereco": "Rua Z" },
12. { "nome": "Zutano", "endereco": "Rua Y" },
13. ]
14. x = collection.insert _ many(lista _ contatos)
15. dados = collection.find()
16. G = nx.DiGraph()
17. for x in dados:
18. G.add _ edge(x["nome"], x["endereco"] )
19. nx.draw _ shell(G, with _ labels=True, node _ size=3000)
```

O resultado é o grafo mostrado na Figura 7.

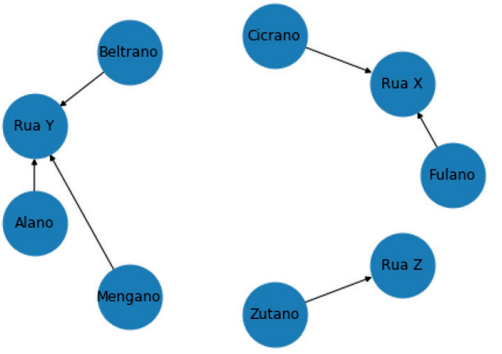

Figura 7. Grafo criado a partir do MongoDB.

Formas de classificação de grafos

Grafos podem ser classificados de acordo com seus vértices e arestas. Você pode levar em consideração a quantidade, a disposição desses elementos e o padrão de ligação formado. Conforme Rosen (2010), vamos começar com algumas definições, listadas as seguir.

- Um **laço** (ou *loop*) é uma aresta que tem origem e destino no mesmo vértice. Se um grafo representa uma rede de computadores, em que cada vértice é um *host* e as arestas são envio de pacotes, quando você utiliza o comando `ping localhost`, você está enviando pacotes cuja origem e cujo destino são o mesmo *host*.
- **Arestas múltiplas** são arestas que possuem o mesmo par de arestas de origem e de destino. Em uma aplicação de transporte e entregas como Uber, por exemplo, você pode ter vários preços diferentes para uma viagem entre duas localidades, dependendo do horário. Grafos que possuem arestas múltiplas são chamados de **multigrafos**.
- Grafos que apresentam laços e arestas múltiplas são chamados de **pseudografos**.
- Um grafo é considerado **simples** quando ele não possui *loops* ou arestas múltiplas.
- Um grafo pode ser definido como **direcionado** se suas arestas representarem a direção da ligação entre dois vértices quaisquer. No Twitter, por exemplo, você pode seguir uma pessoa, mas não necessariamente ela seguirá você de volta. Grafos direcionados também são chamados de **dígrafos**.

O Quadro 1 resume essa terminologia.

Quadro 1. Terminologia dos grafos

Tipo	Arestas	Arestas múltiplas permitidas?	Laços permitidos?
Grafo simples	Não orientadas	Não	Não
Multigrafo	Não orientadas	Sim	Não
Pseudografo	Não orientadas	Sim	Sim
Grafo orientado simples	Orientadas	Não	Não
Multigrafo orientado	Orientadas	Sim	Sim
Grafo misto	Orientadas e não orientadas	Sim	Sim

Fonte: Rosen (2010, p. 592).

Rosen (2010, p. 598–600) ainda nos oferece mais quatro definições importantes:

> DEFINIÇÃO 1:
> Dois vértices *u* e *v* em um grafo não orientado *G* são ditos *adjacentes* (ou *vizinhos*) em *G* se *u* e *v* são extremidades de uma aresta de G. Se *e* estiver associado a {*u*, *v*}, a aresta *e* é dita incidente aos vértices *u* e *v*. Diz-se também que a aresta *e conecta u* e v. Os vértices u e v são chamados de *extremidades* de uma aresta associada a {*u*, *v*}.
> [...]
> DEFINIÇÃO 2:
> O *grau de um vértice de um grafo não orientado* é o número de arestas incidentes a ele, exceto que um laço em um vértice contribui duas vezes ao grau daquele vértice. O grau do vértice *v* é indicado por gr(*v*).
> [...]
> DEFINIÇÃO 3:
> Quando (*u*, *v*) for uma aresta em um grafo *G* com arestas orientadas, dizemos que *u* é *adjacente para v* e *v* é *adjacente a partir de u*. O vértice *u* é dito *vértice inicial* de (*u*, *v*), e *v* é dito *vértice final* ou *terminal* de (*u*, *v*). Os vértices inicial e final de um laço são os mesmos.
> [...]
> DEFINIÇÃO 4:
> Em um grafo com arestas orientadas, o *grau de entrada de um vértice v*, indicado por gr−(*v*), é o número de arestas que tem *v* como seu vértice final. O grau de saída de *v*, indicado por gr+(*v*), é o número de arestas que tem *v* como seu vértice inicial. (Observe que um laço em um vértice contribui 1 tanto para o grau de entrada quanto para o grau de saída desse vértice.)

Vamos, agora, apresentar algumas classes de grafos.

Um grafo é chamado de **completo** se ele é simples (sem laços e arestas múltiplas) e cada vértice é adjacente a todos os outros. Um grafo possui um **ciclo** se ele possui um conjunto de pelo menos três arestas em que o vértice de origem da primeira aresta é o vértice de destino da última aresta. Um grafo que possui um ciclo é chamado de **grafo cíclico**. Se não houver qualquer ciclo no grafo, ele é chamado de **acíclico**.

Fique atento

O conceito de ciclo em um grafo direcionado depende não apenas da existência de arestas entre dois vértices, mas também das direções dessas arestas.

Grafos bipartidos

Um grafo é chamado de **bipartido** quando podemos separar dois conjuntos de vértices desse grafo em que os vértices de um conjunto só se relacionam com os vértices do outro conjunto. Um exemplo desse tipo de grafo é mostrado na Figura 8. Note que os vértices à esquerda só se conectam com os vértices à direita. Uma aplicação para atribuir tarefas a pessoas pode ser modelada a partir desse tipo de grafo: de um lado, você tem o grupo de pessoas e, do outro lado, o conjunto de tarefas a serem realizadas.

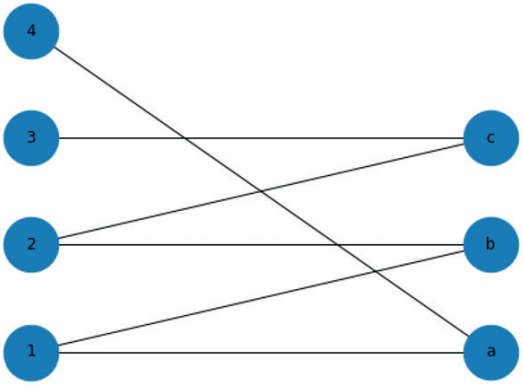

Figura 8. Grafo bipartido.

Caminhos e ciclos em grafos

Um **passeio** (em inglês, *walk*) em um grafo é uma sequência de vértices em que cada vértice é adjacente (vizinho) do outro. Um passeio é dito **fechado** se o primeiro vértice da sequência coincide com o último. Já um **caminho** (*path*) em um grafo é um passeio sem que haja arestas repetidas. Um caminho é dito **simples** se não tem vértices repetidos. Por fim, um **ciclo** é um caminho fechado; ou seja, não tem arestas repetidas e o vértice de origem é o mesmo de destino. Vamos usar o grafo mostrado na Figura 9 para exemplificar esses conceitos.

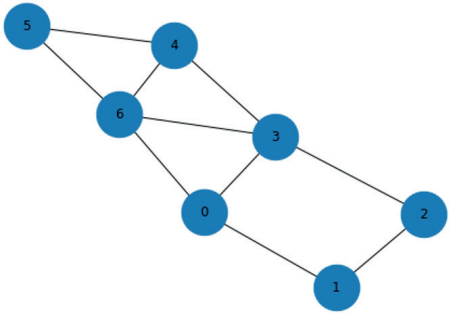

Figura 9. Grafo para exemplificar passeios, caminhos e ciclos.

Você pode ter um passeio contendo a sequência de vértices 0–3–4–5–6. Note que esse passeio é um caminho simples, pois você não repetiu nenhuma aresta e nenhum vértice, mas ele não é fechado (você pode chamá-lo de aberto), pois o primeiro vértice é diferente do último. Se você percorrer os vértices 1–2–3–4–3–2–1, você tem um passeio fechado, porque você sai do vértice 1 e volta para ele. Entretanto, como você repetiu arestas, esse passeio não é considerado um caminho. Por sua vez, a sequência 3–6–4–5–6–0 é um caminho, pois não repete arestas, mas, como passa mais de uma vez no vértice 6, não é considerado simples. Já um ciclo pode ser obtido em: 0–1–2–3–0, 3–4–5–6–3, 0–1–2–3–4–5–6–0 e vários outros. Note que o ciclo 0–3–6–4–5–6–0 não é considerado simples, pois repete o vértice 6.

Caminhos e eulerianos e hamiltonianos

A cidade de Königsberg, na Prússia (agora chamada Kaliningrad e parte da República Russa), era dividida em quatro sessões pelos braços do Rio Pregel.

Essas quatro sessões incluíam as duas regiões das margens do Pregel, a ilha de Kneiphof e a região entre os dois braços do rio. Sete pontes ligavam essas regiões. A Figura 10 descreve essas regiões e as pontes.

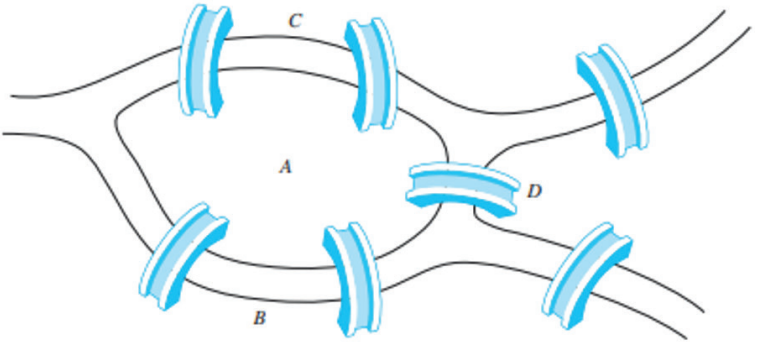

Figura 10. As sete pontes de Königsberg.
Fonte: Rosen (2010, p. 634).

As pessoas da cidade imaginavam se era possível começar um passeio em algum ponto da cidade, atravessar todas as pontes sem cruzar nenhuma ponte duas vezes e retornar ao ponto inicial. O matemático suíço Leonhard Euler resolveu esse problema em 1736. Euler criou um multigrafo em que as quatro regiões são representadas por vértices, e as pontes, por arestas. Esse multigrafo está mostrado na Figura 11. O problema de atravessar todas as pontes sem cruzar nenhuma ponte mais de uma vez pode ser interpretado como: existe um ciclo simples neste multigrafo que contenha todas as arestas?

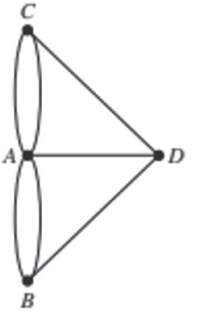

Figura 11. Modelo multigrafo de Königsberg
Fonte: Rosen (2010, p. 634).

Um **ciclo euleriano** em um grafo G é um ciclo simples que contém todas as arestas de G. Um **caminho euleriano** em G é um caminho simples que contém todas as arestas de G. Um caminho simples em um grafo G que passe por todos os vértices exatamente uma vez é chamado de **caminho hamiltoniano** e um ciclo simples em um grafo G que passe pelos vértices exatamente uma vez é chamado de **ciclo hamiltoniano** (ROSEN, 2010).

Note que "euleriano" se preocupa com as arestas, enquanto "hamiltoniano", com vértices. Voltemos ao grafo da figura 8. O caminho 0–1–2–3–4–5–6 é hamiltoniano, pois passa por todos os vértices e uma só vez. O ciclo 0–1–2–3–4–5–6–0 também é hamiltoniano, mas não possui um caminho ou ciclo euleriano, ou seja, não é possível percorrer todas as arestas sem repetir.

Fique atento

Para que um grafo contenha um ciclo ou caminho euleriano, todos os vértices desse grafo devem ter grau par, ou seja, a quantidade de arestas que incidem em cada vértice deve ser um número par. Euler provou que não é possível dar uma volta por Königsberg, voltando ao ponto de partida, cruzando todas as pontes sem repetir (ROSEN, 2010).

Problemas de caminhos mais curtos (ou menores rotas)

Grafos podem ter valores, chamados de **pesos** (*weight*, em inglês), atribuídos a suas arestas. Esses grafos são conhecidos como **grafos ponderados** (*weighted*, em inglês) e podem ser utilizados para representar uma grande quantidade de cenários, como os exemplificados a seguir.

- Um esquema de roteamento de redes na internet, em que cada vértice representa um roteador, e as arestas, os *links* de comunicação entre eles. O peso nas arestas pode representar a velocidade de conexão ou a carga de tráfego naquele *link*.
- Uma rota de entrega de produtos, em que os vértices são os locais de entrega, e as arestas, a distância de entrega entre essas localidades.
- Um sistema de atribuição de tarefas, em que os vértices representam pessoas e tarefas a serem realizadas (grafo bipartido) e as arestas representam o tempo dedicado por uma pessoa na realização de uma tarefa.

Grafos ponderados podem nos levar às seguintes perguntas.

1. Qual é o caminho de menor custo entre dois vértices?
2. Qual caminho permite visitar todos os vértices de um grafo ponderado com menor custo total?

Em 1959, o matemático holandês Edsger Dijkstra criou um algoritmo, apresentado na Figura 12. A partir da escolha de um vértice de origem, o algoritmo de Dijkstra calcula o custo mínimo desse vértice para todos os demais vértices do grafo. O algoritmo pode ser usado sobre grafos orientados (dígrafos) ou não, e admite que todas as arestas possuem pesos não negativos.

procedure *Dijkstra* (G: grafo simples conexo com peso, com todos os pesos positivos)
{G tem vértices $a = v_0, v_1, \cdots, v_n = z$ e pesos $w(v_i, v_j)$ em que $w(v_i, v_j) = \infty$ se (v_i, v_j) não for uma aresta em G}
for i $:= 1$ **to** n
    $L(v_i) := \infty$
$L(a) := 0$
$S := \emptyset$
{os rótulos agora são inicializados de modo que o rótulo de a seja 0 e todos os outros sejam ∞, e S seja o conjunto vazio}
while $z \notin S$
begin
    $u :=$ um vértice não em S, com $L(u)$ mínimo
    $S := S \cup \{u\}$
    for todos os vértices v não em S
        if $L(u) + w(u, v) < L(v)$ **then** $L(v) := L(u) + w(u, v)$
    {isso adiciona um vértice a S com rótulo mínimo e atualiza os rótulos dos vértices que não estão em S}
end {$L(z) =$ comprimento de um caminho mais curto de a a z}

Figura 12. Algoritmo de Dijkstra.
Fonte: Rosen (2010, p. 651).

Problema do caixeiro (vendedor) viajante

Esse problema clássico envolvendo grafos ponderados é enunciado por Rosen (2010, p. 653) da seguinte forma: "Considere o seguinte problema: um caixeiro-viajante quer visitar cada uma de *n* cidades exatamente uma vez e voltar a seu ponto de partida". Se considerarmos cada cidade como um

vértice e as arestas com pesos como as distâncias ou custos de viagem de uma cidade até outra, basicamente o problema se resume a encontrar um ciclo hamiltoniano nesse grafo.

Podemos deixar esse problema mais interessante acrescentando: "[...] um caixeiro-viajante quer visitar cada uma de *n* cidades exatamente uma vez e voltar a seu ponto de partida, **tendo o menor custo possível com suas viagens**". A maneira mais simples de resolver essa variação do problema é examinar todos os possíveis ciclos hamiltonianos do grafo e escolher aquele de comprimento total mínimo. O problema é que essa solução demanda muito tempo computacional para ser implementada. O aumento no número de vértices aumenta exponencialmente o tempo necessário para chegar a uma solução, podendo chegar a valores inviáveis (milhões de anos). Na prática, foram desenvolvidos algoritmos que podem resolver o problema do caixeiro-viajante com até 1.000 vértices utilizando apenas poucos minutos de tempo de computação.

Grafos planares

Um grafo é dito **planar** se ele puder ser desenhado em um plano sem quaisquer arestas se cruzando (em que um cruzamento de arestas é a intersecção de retas ou arcos que as representam em um ponto diferente de sua extremidade comum).

Euler mostrou que a representação planar do grafo divide o plano no mesmo número de regiões. Seja G um grafo planar simples com *e* arestas e *v* vértices, e *r* o número de regiões na representação planar de G. Nesse caso, temos: $r = e - v + 2$. Dessa fórmula, derivam algumas propriedades de grafos planares. Vejamos.

- Se G é um grafo simples conexo e planar, com *e* arestas e *v* vértices, sendo $v \geq 3$, então $e \leq 3v - 6$.
- Se G é um grafo simples conexo e planar, então G tem um vértice de grau menor ou igual a 5.

Na Figura 13, podemos ver gráficos com arestas cruzadas e suas respectivas representações planares.

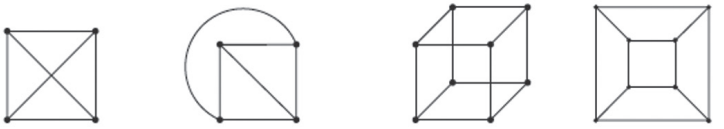

Figura 13. Grafos cruzados e planares.
Fonte: Rosen (2010, p. 658).

Grafos planares têm várias aplicações: no projeto de circuitos integrados, de circuitos impressos, de estradas conectando cidades, de linhas de transmissão de energia elétrica, etc.

Aplicação das técnicas de classificação de grafos em Python

O módulo NetworkX de Python nos permite aplicar os conceitos e as técnicas de classificação de grafos que estudamos até aqui. Iniciaremos propondo um grafo para testes. Criaremos um grafo completo de cinco vértices. O *script* é bem simples:

```
1. import networkx as nx

2. G = nx.complete_graph(5)

3. nx.draw(G, with_labels=True, node_size=2800)
```

O método `complete_graph` gera um grafo completo de *n* vértices. A execução desse *script* gera o resultado mostrado na Figura 14.

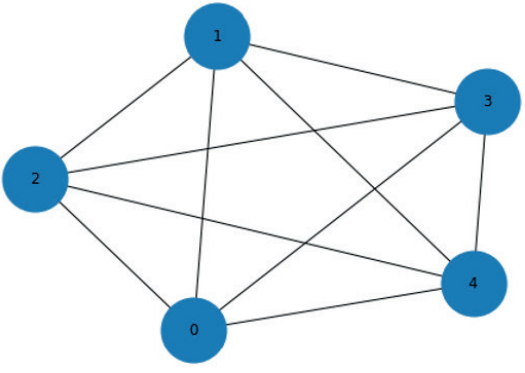

Figura 14. Grafos completo de cinco vértices usado nos testes.

Os atributos `nodes` e `edges` contêm a lista de vértices e de arestas do grafo, respectivamente. Para o grafo de exemplo, retornaria:

```
nodes [0, 1, 2, 3, 4]

edges [(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4),
(2, 3), (2, 4), (3, 4)]
```

O atributo `degree` retorna uma lista de tuplas; nesse caso, `[(0, 4), (1, 4), (2, 4), (3, 4), (4, 4)]`, em que o primeiro elemento é o valor do vértice, e o segundo, seu respectivo grau. Por se tratar de um grafo completo, o grau de todos os vértices é igual a quatro; ou seja, em cada vértice, incidem quatro arestas.

O *script* a seguir gera uma lista com todos os caminhos simples entre dois vértices (no caso, 0 e 3). Para isso, podemos usar o comando `all _ simple _ path`, informando o grafo, o vértice de origem (*source*) e o vértice de destino (*target*):

```
1. import networkx as nx

2. G = nx.complete _ graph(5)

3. for path in nx.all _ simple _ paths(G, source=0, target=3):

4.          print(path)

5. nx.draw(G, with _ labels=True, node _ size=2800)
```

O resultado dessa execução seria:

```
[0, 1, 2, 3]

[0, 1, 2, 4, 3]

[0, 1, 3]

[0, 1, 4, 2, 3]

[0, 1, 4, 3]

[0, 2, 1, 3]

[0, 2, 1, 4, 3]

[0, 2, 3]

[0, 2, 4, 1, 3]
```

```
[0, 2, 4, 3]

[0, 3]

[0, 4, 1, 2, 3]

[0, 4, 1, 3]

[0, 4, 2, 1, 3]

[0, 4, 2, 3]

[0, 4, 3]
```

Para obter a lista de arestas de cada caminho, podemos usar o código a seguir:

```
1. import networkx as nx

2. G = nx.complete _ graph(5)

3. paths = nx.all _ simple _ paths(G, source=0, target=3)

4. for path in map(nx.utils.pairwise, paths):

5.          print(list(path))
```

Sua execução retornaria:

```
[(0, 1), (1, 2), (2, 3)]

[(0, 1), (1, 2), (2, 4), (4, 3)]

[(0, 1), (1, 3)]

[(0, 1), (1, 4), (4, 2), (2, 3)]

[(0, 1), (1, 4), (4, 3)]

[(0, 2), (2, 1), (1, 3)]

[(0, 2), (2, 1), (1, 4), (4, 3)]

[(0, 2), (2, 3)]

[(0, 2), (2, 4), (4, 1), (1, 3)]

[(0, 2), (2, 4), (4, 3)]
```

```
[(0, 3)]

[(0, 4), (4, 1), (1, 2), (2, 3)]

[(0, 4), (4, 1), (1, 3)]

[(0, 4), (4, 2), (2, 1), (1, 3)]

[(0, 4), (4, 2), (2, 3)]

[(0, 4), (4, 3)]
```

O comando `eulerian _ circuit` retorna um circuito euleriano para o grafo desejado, como podemos ver no código a seguir:

```
1. import networkx as nx

2. G = nx.complete _ graph(5)

3. print(nx.is _ eulerian(G))

4. print(list(nx.eulerian _ circuit(G, source=1)))

5. nx.draw(G, with _ labels=True, node _ size=2800)
```

Esse resultado é mostrado na Figura 15. O comando `is _ eulerian` retorna um valor booleano (nesse caso, `True`) se o grafo em questão é euleriano. A lista contém as arestas que formam um circuito euleriano a partir do vértice 1. Em um circuito euleriano, todos os vértices do grafo são percorridos uma única vez no caminho.

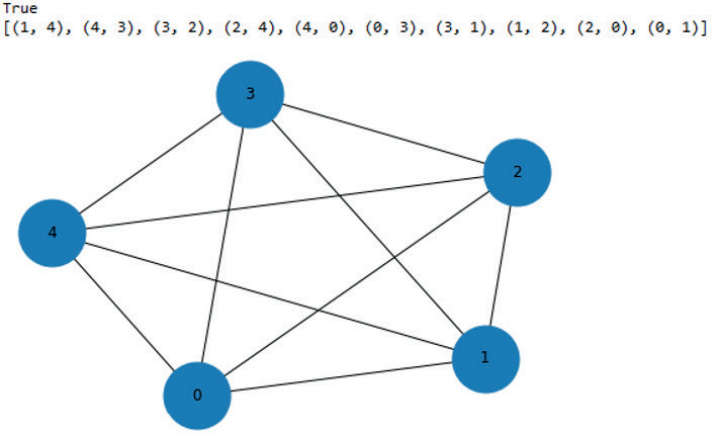

Figura 15. Circuito euleriano para um grafo completo de cinco vértices a partir do vértice 1.

Calculando menores caminhos

O NetworkX contém funções para cálculo de menores caminhos em um grafo. Para fins de teste, vamos considerar o grafo gerado pelo *script*:

```
1. import networkx as nx

2. fh = open("sp.edgelist", "rb")

3. G = nx.read _ edgelist(fh)

4.fh.close()

5. pos=nx.spring _ layout(G)

6. nx.draw(G,pos,with _ labels=True)

7. labels = nx.get _ edge _ attributes(G,'weight')

8. nx.draw _ networkx _ edge _ labels(G,pos,edge _ labels=labels)

9. print(nx.dijkstra _ path(G, '1', '6'))

10. print(nx.dijkstra _ path _ length(G, '1', '6'))
```

Nesse exemplo, estamos definindo a estrutura do grafo por meio de um arquivo de texto chamado de `sp.edgelist`. Dentro desse arquivo, definimos a lista de arestas, com os respectivos pesos (*weights*), que devem ser inseridas no grafo. O arquivo tem o seguinte conteúdo:

```
1 2 {'weight':1}

1 3 {'weight':2}

2 4 {'weight':2}

3 4 {'weight':3}

3 5 {'weight':3}

4 6 {'weight':2}

5 6 {'weight':2}
```

Os dois primeiros valores são o vértice de origem e de destino de uma aresta e o dicionário Python, em seguida, é utilizado para definir os atributos da aresta. Nesse exemplo, definimos apenas o atributo `weight`, que armazena

o peso da aresta em questão. Essa é mais uma opção para definir grafos utilizando o NetworkX.

Nas linhas 9 e 10, estamos aplicando o algoritmo de Dijkstra para encontrar o caminho de menor custo (`dijkstra.path`) entre os vértices de origem (`'1'`) e destino (`'6'`) e imprimir seu custo total (`dijkstra _ path _ length`). O resultado está ilustrado na Figura 16.

```
['1', '2', '4', '6']
5
```

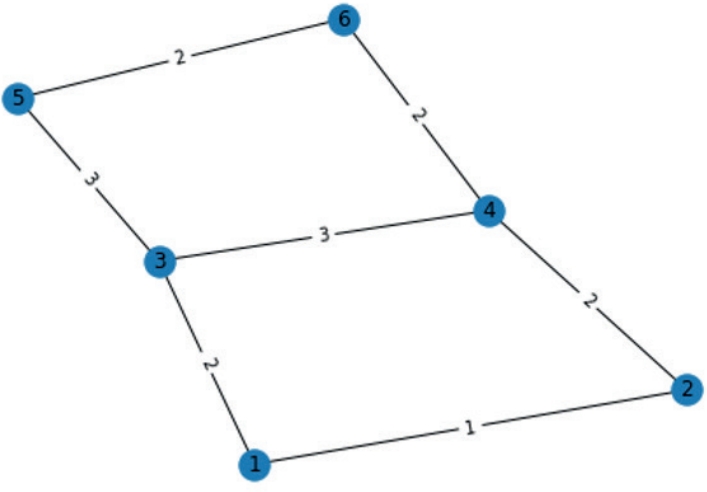

Figura 16. Aplicação do Dijkstra para encontrar o menor caminho entre `'1'` e `'6'`.

O módulo NetworkX é uma ferramenta muito poderosa para ajudar você a manipular grafos. Há centenas de funções muito úteis para atender a uma enorme variedade de aplicações. Não pare por aqui: estude a referência da ferramenta e use o poder dos grafos para potencializar seu *data mining*.

Referências

CHAKRABARTI, D. Graph mining. *In*: SAMMUT, C.; WEBB, G. I. (ed.). *Encyclopedia of machine learning*. Boston: Springer, 2011.

HAGBERG, A. A.; SCHULT, D. A.; SWART, P. J. Exploring network structure, dynamics, and function using NetworkX. *In*: PYTHON IN SCIENCE CONFERENCE, 7., 2008, Pasadena. *Anais eletrônicos* [...]. Disponível em: http://conference.scipy.org/proceedings/SciPy2008/paper_2/full_text.pdf. Acesso em: 15 jan. 2021.

ROSEN, K. H. *Matemática discreta e suas aplicações*. 6. ed. Porto Alegre: AMGH, 2010.

Leituras recomendadas

BATHIA, P. *Data mining and data warehousing*. Cambrige: Cambridge University Press, 2019.

DRABAS, T. *Practical data analysis cookbook*. Birmighan: Packt, 2016.

LAYTON, R. *Learning data mining with Python*. 2nd ed. Birmighan: Packt, 2019.

PLATT, E. L. *Network science with Python and NetworkX quick start guide*. Birmighan: Packt, 2019.

Fique atento

Os *links* para *sites* da *web* fornecidos neste capítulo foram todos testados, e seu funcionamento foi comprovado no momento da publicação do material. No entanto, a rede é extremamente dinâmica; suas páginas estão constantemente mudando de local e conteúdo. Assim, os editores declaram não ter qualquer responsabilidade sobre qualidade, precisão ou integralidade das informações referidas em tais *links*.

Conteúdo:

SAGAH
SOLUÇÕES EDUCACIONAIS INTEGRADAS