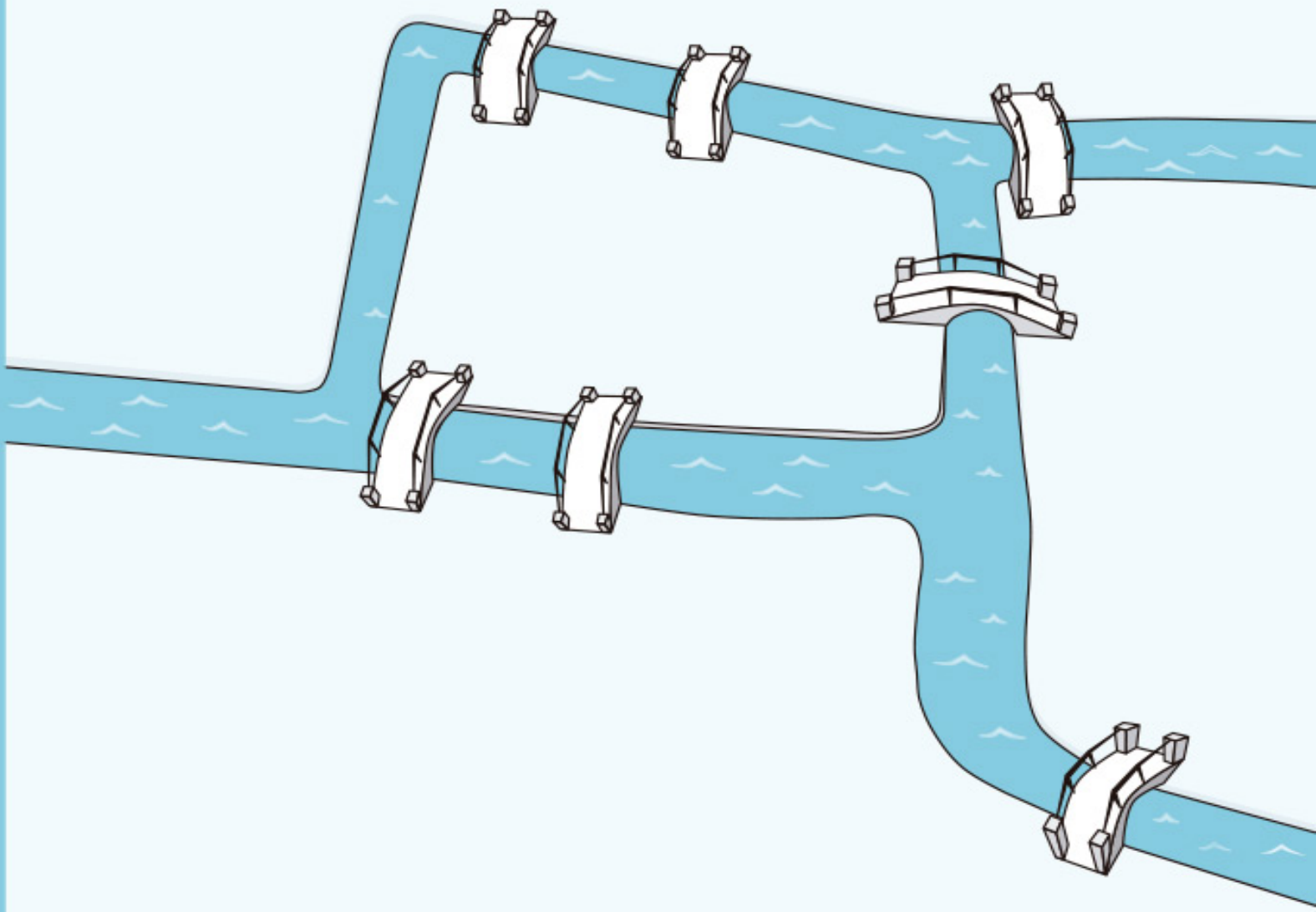
Teoria dos Grafos

Uma abordagem prática em Java



Casa do Código

JOÃO PAULO MAIDA

Sumário

ISBN

Impresso e PDF: 978-65-86110-51-7

EPUB: 978-65-86110-50-0

MOBI: 978-65-86110-52-4

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Sobre o autor

Olá! Maidão, Maidinha, Zé, Peixe, Peixão etc. Esses são meus apelidos, mas meu nome verdadeiro é João Paulo (não sei por que meu pai me chama de Zé). Geralmente me apresento como Maida, meu sobrenome, porque sempre existe um outro João por aí, Maida é mais difícil de encontrar.

Comecei em TI com quinze anos, no curso técnico. Sempre gostei de tecnologia, ciências no geral e do espaço. Um bom nerd de carteirinha. Terminei minha graduação em Ciência da Computação em 2012 na finada UniverCidade - Universidade da Cidade, com diploma expedido pela Universidade Veiga de Almeida - RJ. Em 2014, comecei meu MBA em Engenharia de Software pela Universidade Federal do Rio de Janeiro - UFRJ, terminando em 2016.

Quando o assunto é trabalho, comecei cedo. Em 2008, já trabalhava com TI, dava aulas de informática básica em um laboratório comunitário. Depois fui para área de infraestrutura onde fiquei até 2010. A partir desse ano até 2014 trabalhei com desenvolvimento de software, foram anos de trabalho árduo.

De 2014 até 2018, eu continuei trabalhando com desenvolvimento, mas não da perspectiva de um desenvolvedor, mas sim de um analista de qualidade/arquiteto. Foi uma experiência interessante e que me abriu outras portas. Em 2018, voltei para área de desenvolvimento para sair novamente em 2019 para a tão sonhada área de arquitetura e é onde estou atualmente, e feliz.

Sou muito apaixonado pelo que faço e gosto muito de estudar. Na verdade, eu curto aprender qualquer assunto. Como parte dessa fome insaciável por conhecimento, criei um blog no qual escrevo sobre os mais diversos assuntos relacionados à tecnologia (https://precisoestudarsempre.blogspot.com/).

Estou no Twitter (@jpmaida) e no LinkedIn (João Paulo Maida). Meu e-mail é `jpmaida@gmail.com`. É um prazer lhe conhecer.

Agradecimentos

Agradecer aos envolvidos nesta obra é algo muito complicado porque a lista é longa. Então, fui obrigado a dar uma resumida para atingir a todos que estiveram comigo durante esta jornada. Se o seu nome não está relacionado diretamente, não fique triste pois saiba que você está no meu coração.

Agradeço primeiramente a Deus por permitir que eu ande nesta terra de cabeça erguida e em plena condição de meus pensamentos. Ele já me deu as ferramentas necessárias para que eu possa ser um vencedor. E a Odin (sim, gosto de pensar que sou viking) por me dar as forças para superar todos os meus desafios.

À Vivian Matsui, minha editora, o meu muito obrigado por ter tido toda a paciência do mundo com a minha demora em escrever os capítulos e pelas dicas dadas.

À minha avó Esther Maida Gonçalves, que nos deixou em dezembro de 2018, o meu muito obrigado pela sua preocupação com meu bem-estar e com esta obra, pelos conselhos, por vibrar a cada vitória minha. Você é imortal em meus pensamentos.

Ao meu cão Tigrão, por ter trazido alegria à minha vida durante seus dezessete anos de existência e por mostrar que o amor não possui fronteiras. Ainda nos encontraremos.

Aos meus pais Fernando e Marcia por tudo que tenho e tive até hoje. Sem vocês, nada disso seria possível. Obrigado por estarem ao meu lado nas vitórias e derrotas. Amo muito vocês.

À minha esposa Carina das Flores, agradeço por todo o amor e compreensão durante a construção deste trabalho. Sem seus conselhos não teria conseguido. Te amo com todo o meu coração.

Ao professor Orlando Bernardo Filho da UERJ, que infelizmente foi desta para uma melhor em outubro de 2019, o meu muito obrigado por ter sido meu mestre e aquele que primeiro me apresentou a Teoria dos Grafos. Posso dizer que tive o privilégio de ser seu aluno. O senhor faz falta.

Ao professor Heraldo Luís da UFRJ por ter me guiado e me apresentado muitos dos algoritmos que debato nesta obra.

Ao professor Cláudio Esperança da UFRJ por ter um coração enorme e ter me ajudado na revisão técnica desta obra. Sem o senhor não conseguiria chegar aonde cheguei.

A todos os meus familiares e amigos, o meu muito obrigado por terem acreditado em mim, pelos estímulos e votos de felicidade feitos. Sem vocês a vida seria chata e sem sentido.

A Felipe Barelli, o meu muito obrigado por ter aberto uma porta para mim em direção a esta oportunidade.

Sobre o livro

Esta obra consiste na reunião de todos os fundamentos que julguei interessantes na área da Teoria dos Grafos, voltada para um público ligado à área de desenvolvimento de software, sejam estudantes, profissionais ou curiosos. Entretanto, para que sua experiência seja a melhor de todas, uma boa noção de Lógica de Programação, Programação Orientada a Objetos e um pouco de vivência com a linguagem Java são necessárias, porque o livro desenvolve junto com o leitor ou a leitora uma aplicação Java no qual emprega toda a teoria vista.

Caso você não possua habilidades com programação, ou possua mas com uma linguagem diferente de Java, não se preocupe. É possível aproveitar toda a parte teórica desenvolvida no livro no primeiro caso e, para o segundo, é sempre bom frisar que nada impede de transpor o conhecimento obtido aqui para uma outra linguagem de programação.

Uma vez que dê início à sua leitura, seu primeiro encontro será com toda a base histórica e teórica que acompanhará você pelo livro, formada pela criação da Teoria dos Grafos e suas diversas aplicações ao longo da história, definições de grafo, vértice, aresta, suas funções e características. Tudo isto é englobado nos capítulos 1 e 2.

No capítulo 3, são vistas as diferentes formas de representação de conexões entre vértices e arestas e suas respectivas características. Esse capítulo serve de suporte para os métodos de buscas abordados no capítulo 4.

No capítulo 5, você se encontra com novas representações para grafos, as árvores, junto com suas características, funções, curiosidades e procedimentos.

O capítulo 6 adiciona novas características aos grafos já conhecidos e mostra quais pontos no projeto desenvolvido no livro precisam ser

modificados para suportar essa inclusão. O capítulo 7 utiliza de toda a teoria vista nos capítulos anteriores e apresenta algoritmos utilitários que respondem a questões do cotidiano.

O capítulo 8 possui uma abordagem que eu chamo de *ping-pong*. Foi todo estruturado para mostrar pedaços de código e esperar que você os implemente para assim avançar na construção que finaliza a aplicação.

O capítulo 9 conclui a obra com reflexões e deixa uma mensagem para você, como uma forma de expressar meu carinho por ter adquirido meu livro.

O intervalo entre os capítulos 2 até o 8 compreende toda a parte técnica referente a Teoria dos Grafos e à aplicação desenvolvida. Aproveite com sabedoria.

A aplicação completa, assim como dividida por capítulos, pode ser encontrada no repositório GitHub através do link https://github.com/jpmaida/livro-algoritmos-grafos-java. Utilize-o como o seu guia em casos de dúvidas.

Capítulo 1
O início de tudo

1.1 Antes de tudo, um pouco de história

Desde do início da nossa existência até hoje, foram diversas as vezes em que ficamos maravilhados com fenômenos da natureza, e incontáveis foram nossas tentativas de entendê-las. Em nossa perseguição por conhecimento, passamos por diversas dificuldades, mas sempre visando colher melhores frutos no futuro. Contudo, vamos pensar em quando isso começou? Lembre-se dos egípcios, por exemplo.

Há quase 6 mil anos, este incrível povo realizou magníficos feitos na área da construção e da engenharia com recursos extremamente simples, como areia, argila e palha. Então, a pergunta despertada em nós é: como? Como, a partir do nada, eles construíram tudo? É desnecessário dizer que seu domínio matemático era imenso. Sua grandeza era tão monstruosa que o que foi construído milênios atrás ainda está de pé, recebendo visitas todo o ano e revelando segredos ainda escondidos. A cada momento novas descobertas são feitas, assim desvendamos o passado. Exemplos como esse se espalham por nossa história.

Avançando um pouco mais no tempo, havia um boato urbano em 1736 de que era impossível cruzar todas as pontes de um lugar conhecido como Königsberg. A topologia do local era constituída de sete pontes que ligavam as duas ilhas entre elas e ao restante do território, conforme a figura a seguir (Harary, Frank; 1970):

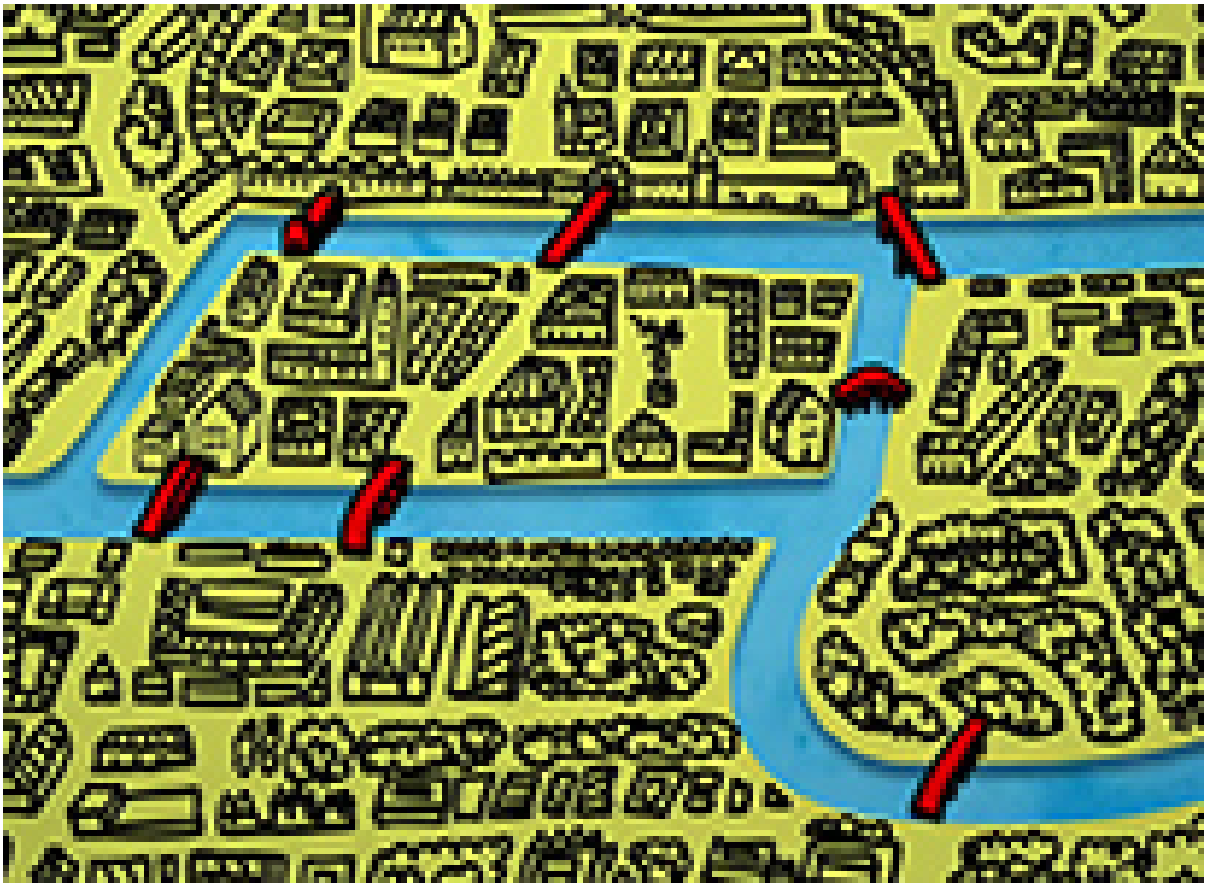


Figura 1.1: As sete pontes de Königsberg. Fonte: http://www.im-uff.mat.br/festival/cineclube/ted-konigsberg.jpg

O desafio consistia em atravessar todas as pontes, passando por cada uma somente uma vez, e ter como ponto de chegada o mesmo ponto de partida. Obviamente, é possível tentar resolver esse problema de forma empírica, na base da tentativa e erro, mas você notará que todas suas tentativas resultarão em falhas (Harary, Frank; 1970). Tal se deve ao fato de ser realmente impossível resolver o boato nas condições apresentadas e a primeira pessoa que apresentou tal conclusão foi o grande Leonhard Euler (1707-1783). Assim, nascia o primeiro grafo conhecido.

Este é o legado de Euler para a humanidade, a Teoria dos Grafos, e seu valor é inestimável. Através da simples comprovação da inexequibilidade do problema, foi originado um novo ramo da matemática, com milhões de possibilidades de aplicação.

Entretanto, uma pergunta ainda não foi respondida: como ele conseguiu chegar a tal conclusão? Euler teve a brilhante ideia de substituir cada terreno por um ponto e cada ponte por uma linha que conecta dois pontos. O resultado obtido foi o grafo mostrado pela imagem a seguir. Cada ponto e linha podem ganhar um rótulo para fins de identificação com as áreas da região. Com isso, Euler conseguiu demonstrar que o grafo não pode ser atravessado da forma proposta pelo boato (Harary, Frank; 1970).

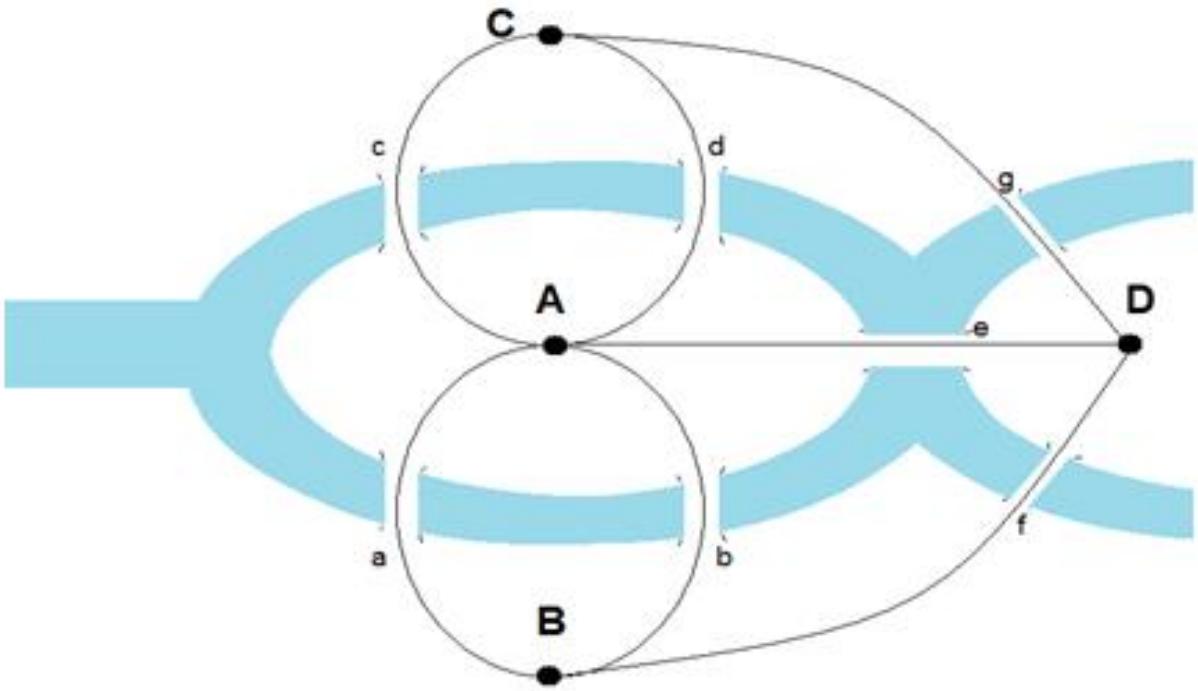


Figura 1.2: O grafo proposto por Euler. Fonte: https://bit.ly/2L6j5ww

Com uma simples pesquisa na internet, é possível encontrar diversos exemplares distintos do grafo do Problema das Sete Pontes de Königsberg. Não se desespere, porque todos representam a mesma coisa. Segundo Bondy e Murty, não há uma única forma de se desenhar um grafo. Alguns podem ter rótulos somente em suas linhas, outros somente em seus pontos e outros nos dois. Inclusive a disposição dos elementos também não tem importância. Um pouco mais acima ou pouco mais abaixo não faz

diferença. Ao contrário, é normal. Na verdade, um grafo só possui esse nome porque ele pode ser representado graficamente, é essa representação que nos ajuda a entender muitas de suas propriedades. No fim das contas, o que realmente importa são as conexões feitas (Bondy, J. A.; Murty, U. S. R.; 1976).

Mais uma vez, uma reflexão é necessária. Note que na solução de um simples boato estava contido algo absolutamente incrível, complexo e completamente novo aos olhos das pessoas daquela época. Tal fenômeno se repete constantemente na natureza. Muitas vezes conseguimos extrair respostas complexas a partir de eventos simples, como o voar de uma abelha. As consequências da criação de Euler se estendem até hoje. Vivemos em um mundo em que informação é poder e não basta somente possuí-la, mas sim processá-la. A teoria dos grafos chega a nossas mãos como uma ferramenta para abstrair problemas reais e levá-los para uma outra dimensão. A partir de lá e somente de lá conseguimos extrair respostas outrora inimagináveis.

1.2 Na máquina do tempo, voltando para o presente

Se você, assim como eu, também nasceu na década de 90 deve estar bastante acostumado(a) a lidar com tecnologias como smartphones, GPS, jogos eletrônicos, redes sociais, programas de compactação de dados, entre outros. Contudo, há trinta anos, quem pensaria que uma máquina que cabe na palma da mão conseguiria nos dizer um caminho completo de uma cidade à outra? E não somente fornecer um caminho qualquer, mas sim o melhor caminho, dado algum critério (tempo, periculosidade, quantidade de postos de gasolina etc.) estabelecido pelo próprio usuário. Em todos esses casos, a teoria dos grafos está presente e, consequentemente, a matemática. A partir de alguns pontos e linhas no espaço,

conseguimos extrair resultados fantásticos. Porém, o que define a necessidade do uso de um grafo?

A resposta para essa pergunta pode parecer difícil, mas não é. O que vai definir a verdadeira necessidade é o problema em questão. Caso seus elementos possuam alguma relação entre si, então você já tem conexões e, assim, um grafo. Pense em um mapa das suas amizades da sua rede social favorita, assim como o LinkedIn Maps fazia. Atualmente foi descontinuado, infelizmente, mas tive a sorte de gerar o meu antes que fosse tarde demais. Esse mapa mostrava todas as minhas amizades da época e as relacionava, assim como relacionava amizades de meus amigos, os quais tinham-me como amigo em comum.

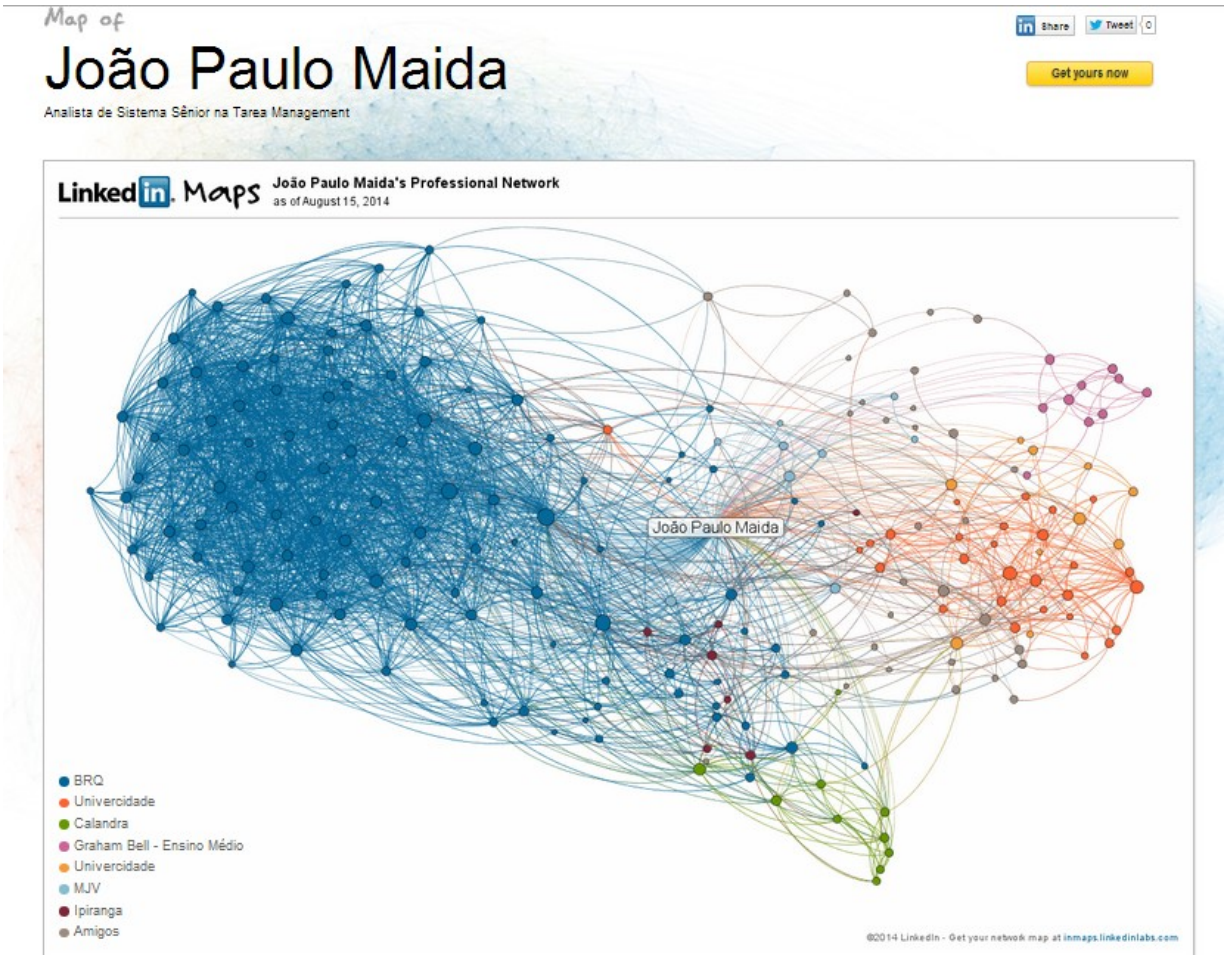


Figura 1.3: Meu mapa de amizades do LinkedIn

Note como é denso e colorido. Quem seriam os pontos desse grafo? Eu e meus amigos. E quem seriam as linhas? Minhas amizades e as amizades de meus amigos. Pronto! Já possuímos um grafo que representa meu mapa de amizades. A partir dele podemos realizar qualquer tipo de processamento, como buscas ou traçar o caminho mais curto de um amigo a outro. Mais uma vez nos deparamos com uma situação mundana que possui intrinsecamente diversos desafios matemáticos.

Não me estenderei muito mais neste capítulo até porque a verdadeira diversão está para começar. Acredito que ao ler este parágrafo sua boca já está adoçada com a curiosidade de descobrir o quão funda é a toca do coelho. Pense nas infinitas aplicações já descobertas e as que ainda estão por vir. Nos capítulos à frente, cobriremos conceitos básicos, representações, mais aplicações no mundo real e algoritmos de processamento.

1.3 Conclusão

Realmente, a natureza é algo formidável. Muitas vezes me pego pensando em como somos pequenos como seres humanos perto da imensidão do cosmos. Ainda há muito para se aprender e muito para ensinar. Um grande exemplo dessa grandiosidade que nos cerca é a Teoria dos Grafos, criada a partir de um simples boato urbano.

Este ramo da matemática disponibilizou um meio de se analisar problemas de uma forma nunca vista antes, por meio das relações entre seus elementos. A consequência disso são as inúmeras aplicações e vantagens obtidas que fazem parte da nossa vida, de tal forma que não conseguimos viver mais sem elas. E se formos parar realmente para pensar, quantas foram as descobertas que aconteceram do mesmo modo? Muitas, mas em todas existe um elemento que sempre se repete: a curiosidade do ser humano, ela é

combustível para nossas respostas e para o futuro. Somente através dela chegamos a um nível de abstração tão alto que nos possibilita transformar a relação entre elementos de um determinado contexto em algo não abstrato.

Resumo

- A Teoria dos Grafos está mais presente na nossa vida do que pensamos. Aplicativos de GPS, jogos, programas de compactação de dados, redes sociais são exemplos de aplicações deste ramo da matemática.
- O problema das sete pontes de Königsberg consistia em um boato urbano que afirmava que era impossível cruzar todas as pontes de Königsberg chegando por onde partiu sem repetir nenhuma ponte.
- Através da solução que mostra que o problema das pontes de Königsberg é impossível, Leonhard Euler descobriu uma nova ramificação da matemática, a Teoria dos Grafos.
- Não existe uma forma única de desenhar um grafo. É possível encontrar diversos grafos representados de forma diferente, mas com os mesmos pontos e linhas. O que realmente importa são as conexões entre os pontos e as linhas.
- Um grafo é composto por nós e arestas. Os nós podem levar rótulos para fins de identificação assim como as arestas.
- A necessidade de criação de um grafo é determinada pelo problema que ele representa. Caso o problema apresente elementos que tenham algum tipo de relacionamento comum, então a partir deles podem ser derivados pontos e conexões entre eles.

Capítulo 2
Grafos

Neste capítulo, abordaremos conceitos que formarão a nossa base de conhecimento. Após entender as diversas definições e características que os grafos podem oferecer, veremos como é possível transformar tudo o que foi visto em algo realmente funcional. Essa representação funcional nos acompanhará por todo o livro e em cada capítulo sofrerá modificações que contemplarão os novos conceitos. É importante deixar claro que seu estado final sempre será completo, ou seja, não deixaremos pontos soltos de um capítulo para o outro. Finalmente, consolidaremos todo o conteúdo visto em uma conclusão para formar o resumo com os principais pontos vistos.

Bem-vindo(a) ao início da jornada.

2.1 O que de fato é um grafo?

Definições são importantes e isto é um fato. Mas ao mesmo tempo que definições são importantes, acordos também são. Então precisaremos abordar os dois. Vamos definir muitos conceitos que nos seguirão até o fim da jornada e faremos acordos para que não haja confusão.

Já vimos que um grafo é uma abstração matemática de problemas reais composta de pontos e linhas, até aí nenhuma novidade. Mas, descendo um degrauzinho, em um linguajar mais técnico, o que de fato é um grafo?

Um grafo G(V,E) é uma estrutura composta por dois conjuntos denominados V e E, onde V representa o conjunto de vértices e E, o de arestas (Even, Shimon; 1979). Entretanto, o que seriam vértices

e arestas? É denominado nó ou vértice tudo aquilo que representa um elemento de um problema, e aresta, tudo que representa uma relação entre dois vértices. É muito comum encontrar na literatura ou em sites diversos termos para representar vértices e arestas, mas não se preocupe, são apenas sinônimos.

Diversos autores também definem um grafo como uma tripla de elementos, na qual os dois primeiros são os conjuntos já citados e o terceiro é uma função de incidência ? (Bondy, J. A.; Murty, U. S. R.; 1976). Entenda esta função como uma ferramenta que tem o poder de representar a conexão de uma aresta e não se assuste com a letra grega minúscula Psi, ela não morde. Não utilizaremos esta abordagem por motivo de simplificação do conceito, porém nada impede que conheçamos uma outra forma de representar o que já conhecemos.

Eis aqui nossa primeira definição:

Definição:

Um grafo G(V,E) é formado pelos conjuntos V e E, onde:

V = {v1,v2,v3,.. }

E = {e1,e2,e3,...}

Agora, com ela, podemos realizar o nosso primeiro acordo:

Acordo:

Os pontos de um grafo serão chamados de vértices, e as linhas, de arestas.

Aplicando esses conceitos ao grafo G a seguir, como seria possível representá-lo de forma matemática?

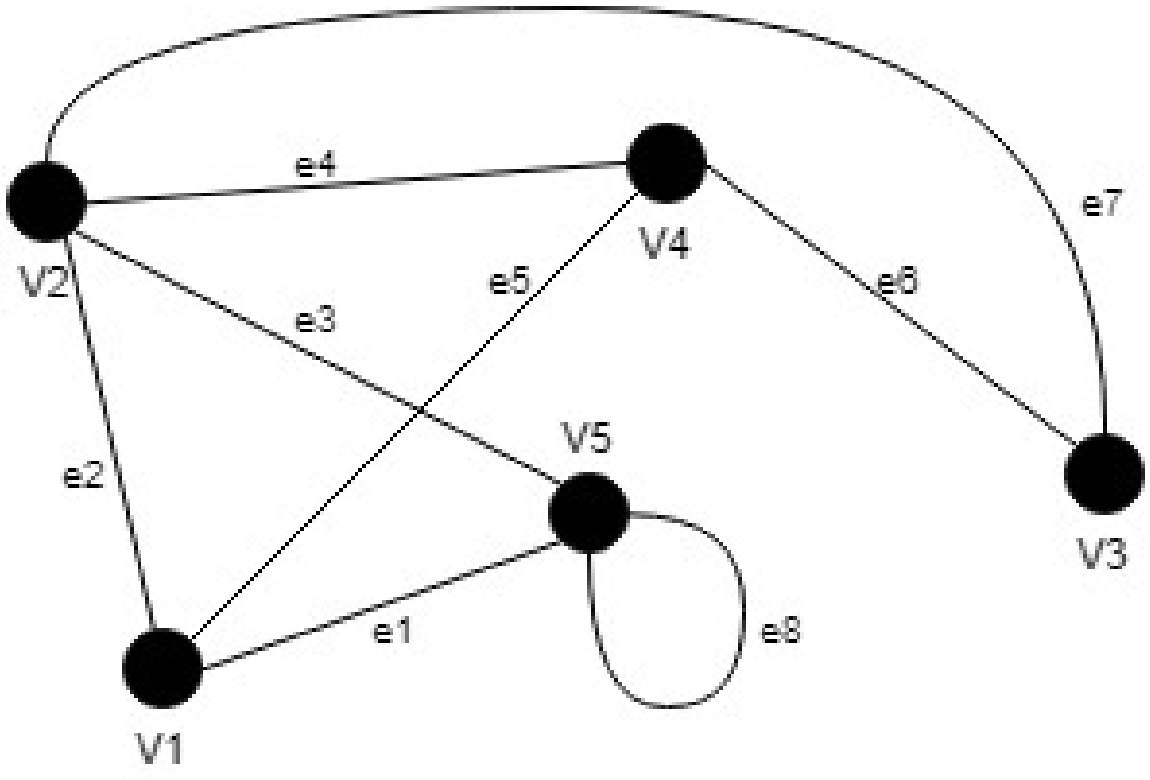


Figura 2.1: Grafo G não orientado

Primeiro, precisamos identificar seus vértices, representados por bolinhas na imagem, e adicionar todas as encontradas ao conjunto `V`. Logo, este conjunto será `V = {v1,v2,v3,v4,v5}`. Em seguida, contabilizamos todas as arestas no conjunto `E` para chegar à seguinte conclusão: `E = {e1,e2,e3,e4,e5,e6,e7,e8}`. E assim, o grafo `G(V,E)` é finalmente composto pelos seus dois conjuntos.

Cada aresta possui duas pontas, sendo que uma está ligada a um vértice e a outra está ligada a um outro vértice. Porém, existem casos em que um vértice pode ter uma aresta que começa e termina nele. Nestes casos, dizemos que a aresta é recursiva ou em *self-loop* (Even, Shimon; 1979). Tal situação é exemplificada pela aresta `e8` do nosso exemplo.

A sequência de conexões entre vértices e arestas organizados de forma alternada forma um caminho, como podemos observar. Portanto, podemos afirmar que um dos infinitos caminhos possíveis

a partir do vértice $v1$ é passar por $v2$ e $v4$ até o vértice $v3$, situado no outro extremo. Podemos representar este caminho P da seguinte forma (Even, Shimon; 1979):

```
P : v1-e2-v2-e4-v4-e6-v3
```

Qual é a importância de um caminho? É imensa. Um caminho não é somente uma simples sequência, ele constitui uma forma sólida de representação de um trajeto realizado. Pense em um sistema de logística de entregas de uma empresa. Todo dia rotas são montadas, ou seja, tudo se resume aos caminhos feitos, todo o valor de mercado da empresa está aí. A empresa que conseguir formar as melhores rotas a fim de realizar entregas mais rápidas ganhará, consequentemente, mais espaço em mercado. Todo o trabalho da construção de rotas só é possível através da análise de caminhos.

Definição

Um caminho é uma sequência de vértices e arestas organizados de forma alternada (Even, Shimon; 1979).

Em casos em que nenhum vértice ou aresta se repete, o caminho é definido como simples, assim como P. Por outro lado, para que um caminho possa ser considerado um *circuito*, o seu vértice inicial deve ser igual ao seu vértice final. Obviamente, este não é o caso de P. Um caminho é dito vazio quando o seu comprimento é zero. O comprimento de um caminho é definido pela quantidade de arestas que o compõe (Even, Shimon; 1979). Logo, para o nosso exemplo, o comprimento 1, do caminho P, o qual pertence ao grafo G, é 3.

Os fatos apresentados no último parágrafo podem tê-lo deixado pensativo da seguinte forma: já que um circuito é definido por um caminho que tem seus vértices iniciais e finais iguais, um vértice com uma aresta recursiva seria considerado um circuito?

Olhando para o vértice `v5` e a aresta `e8`, seu caminho `P` seria definido da seguinte maneira:

```
P : v5-e8-v5
```

Tal caminho forma um circuito? Evidentemente. Se nos atermos à teoria, de fato, `P` inicia e termina no mesmo vértice. Mas será ele somente um circuito? Um circuito é definido como simples quando nenhum outro vértice se repete além de seu inicial e final. Logo, o circuito em questão é simples e possui comprimento igual a um. Mas caso tivéssemos um outro circuito que começasse e terminasse no mesmo vértice `v5`, e esse intervalo possuísse outros vértices, não seria considerado simples.

DEFINIÇÃO

Caminho simples: é todo aquele em que nenhum vértice, e consequentemente nenhuma aresta, se repete.

Circuito: é todo aquele caminho em que tanto o vértice inicial quanto o final são os mesmos.

Circuito simples: é todo aquele circuito em que nenhum vértice se repete, exceto o inicial e o final.

Os vértices analisados até agora possuem três, no máximo quatro arestas ligadas a eles. Determinamos isso através de uma simples contagem, mas e se precisássemos de uma ferramenta que faça essa tarefa por nós? Para isso, entender o grau de um vértice se torna necessário.

O grau de um vértice é determinado pela quantidade de arestas conectadas a ele (Even, Shimon; 1979). Sempre que quisermos nos referenciar a essa ferramenta utilizaremos a nomenclatura `d(v)`, onde `d` representa a funcionalidade que calcula o grau, e `v` um vértice qualquer. Logo, para o vértice `v4` do grafo `G` acima, seu respectivo grau é representado por `d(v4) = 3`, pois existem três

arestas conectadas a ele, são elas: `e4, e5, e6` . Não subestime esta função pela sua simplicidade, pois ela possui um grande valor em algoritmos mais complexos que veremos à frente.

DEFINIÇÃO

O grau de um vértice é definido pela quantidade de arestas conectadas a ele.

Em todos os conceitos que já abordamos, não paramos para pensar ainda se uma aresta pode possuir ou não uma direção. Mas partindo do pressuposto de que ela pode, qual seria o impacto? Imenso, evidentemente. Antes, quando discutíamos sobre grafos não orientados, ou seja, aqueles cujas arestas não possuem direção, era possível ir de um vértice a outro e vice-versa sem problemas. Contudo, quando inserimos uma direção em uma aresta, só um trajeto poderá ser feito, aquele definido pela sua orientação.

Note a imensa diferença que o grafo `G` da figura anterior sofre quando adicionamos direções em suas arestas.

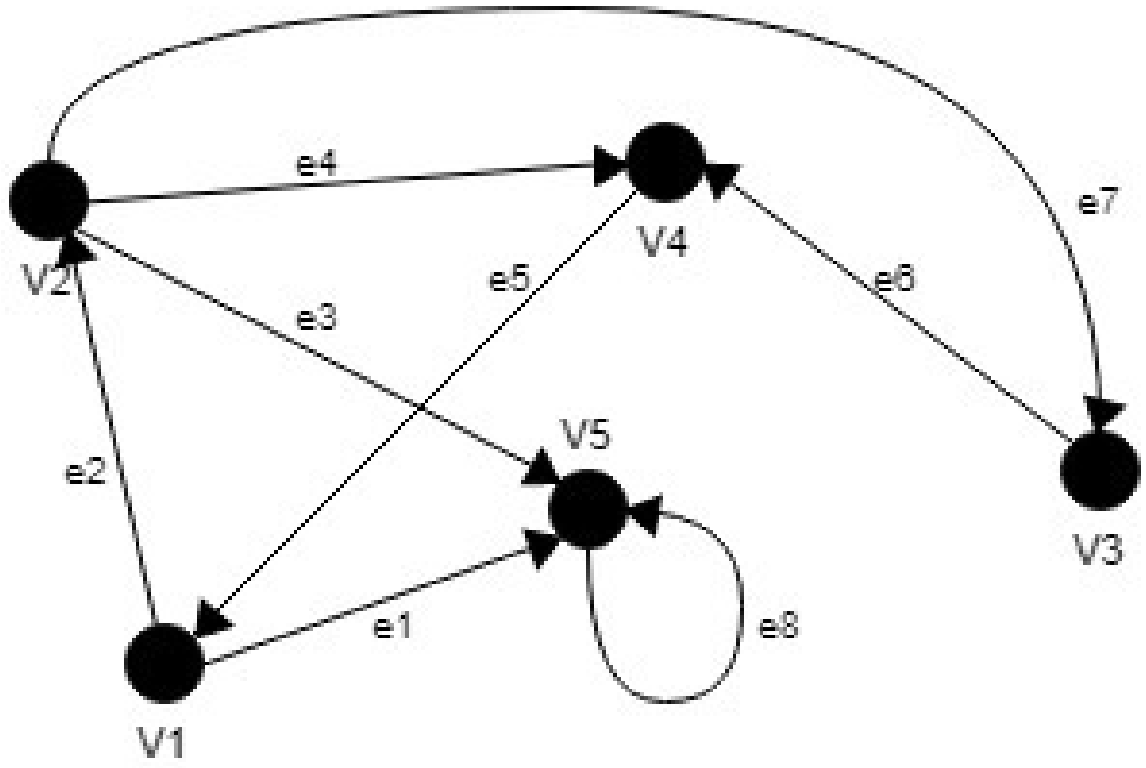


Figura 2.2: Grafo G orientado

Agora não é mais possível ir de $v1$ a $v3$ passando somente por $v4$. O único caminho viável é indo por $v2$ direto a $v3$. Esta diferença pode parecer sutil, visto que ainda foi possível atingir o local de destino. Contudo, ela perde a sutileza e se torna muito evidente quando, por exemplo, mudamos no GPS o tipo de trajeto que vamos fazer. Ainda no mesmo exemplo, se você deseja sair de sua casa, pegar seu carro e ir ao shopping mais próximo, o GPS calculará um determinado caminho. Entretanto, tente para a mesma origem e destino mudar o tipo de trajeto para "a pé". O caminho calculado será outro, completamente diferente.

A diferença entre os trajetos se torna óbvia com um pouco de reflexão. O que devemos respeitar em um passeio a carro que podemos não respeitar em um passeio a pé? As direções. Se estou de carro em uma via que só tem uma direção, devo respeitá-la. Caso seja da minha vontade ir na direção contrária, devo procurar

algum tipo de retorno e a partir daí traçar um novo caminho. Estando a pé, tomo o caminho que quero e mudo quando eu quiser.

A adição desta nova característica muda os valores dos graus dos vértices, pois para grafos orientados contamos somente as conexões incidentes ao vértice, ou seja, as arestas que apontam para ele. Então, para este novo grafo G orientado, o `d(v4)` que antes era 3 será agora 2, pois somente `e4` e `e6` são incidentes a `v4`.

Definição

Grafos orientados ou dígrafos, como também são conhecidos, são similares em estruturas e propriedades a grafos não orientados, exceto que suas arestas possuem direções.

Com isso, já temos teoria o suficiente para sair do campo das ideias, onde discutimos e entendemos conceitos, e ir para onde o verdadeiro jogo começa. Então, pegue o seu café e vamos lá.

2.2 Saindo da teoria, entrando na prática

Vamos supor que você trabalha em uma empresa que possui muitas filiais por todo o país e ocupa o cargo de analista de sistemas. Você é bem reconhecido pelo que faz, gosta bastante de ajudar e cooperar. Um certo dia notou que a empresa recebe e envia muitos malotes de documentos diariamente por correio e que às vezes alguns atrasos eram relatados porque alguém tinha mandado um malote por um caminho mais longo do que deveria. Então, ouvindo todos esses rumores, foi checar nas diversas áreas afetadas se não existia um serviço de logística forte atuando neste problema e dando opções aos usuários. Neste exato momento, você encontrou a brecha que estava procurando para ajudar pois obviamente pensou

que com sua supermente avançada poderia resolver esse problema utilizando a Teoria dos Grafos e receber aquele aumento há tanto tempo sonhado.

Nesta seção, iniciaremos a construção da aplicação que será a solução para o seu problema e ela nos acompanhará por todo o livro. O primeiro passo que precisamos dar é buscar mais informações sobre o problema. Até agora sabemos que a empresa em que você trabalha recebe e envia esses malotes com uma frequência diária, existem várias filiais em várias cidades, os envios e recebimentos são feitos via correio e não sabemos avaliar ainda as direções dos trajetos de envio e qual trajeto é o melhor. Acredito que com o andar da carruagem mais informações virão ao nosso conhecimento.

O que podemos fazer nesse momento é identificar os elementos do problema e seus relacionamentos. A esta altura já deve ser claro para você que as filiais junto com a central são os elementos e os rotas de envio entre eles são os relacionamentos. Tendo isso em vista, chegou ao seu conhecimento a existência de uma relação escrita de todas essas rotas, com suas abreviações para os pontos de origem e destino. Para não perder tempo, você já fez a solicitação para o departamento responsável e com ele em mãos começou a análise.

Rio de Janeiro	RJ	SP, BH, PT, PA
São Paulo	SP	RJ,BH,OS,SV,CR,PA
Belo Horizonte	BH	RJ,SP
Petrópolis	PT	RJ
Osasco	OS	SP
Salvador	SV	SP,PA
Curitiba	CR	SP,PA
Porto Alegre	PA	RJ,SP,SV,CR

Figura 2.3: A relação entre cidades.

Essa relação consiste em uma tabela com três colunas, sendo que a primeira representa a localidade da filial/sede, a segunda representa abreviação dessa localização e a terceira, suas conexões com as outras filiais. Portanto, olhando para a figura podemos concluir que a filial Rio de Janeiro (RJ) envia e recebe documentos para as filiais

São Paulo (SP), Belo Horizonte (BH), Petrópolis (PT) e Porto Alegre (PA).

De acordo com o que já aprendemos, sabemos que os elementos de um problema se tornam vértices e seus relacionamentos se tornam as arestas de um grafo. Então, aplicando este conceito, teremos o grafo `G` a seguir.

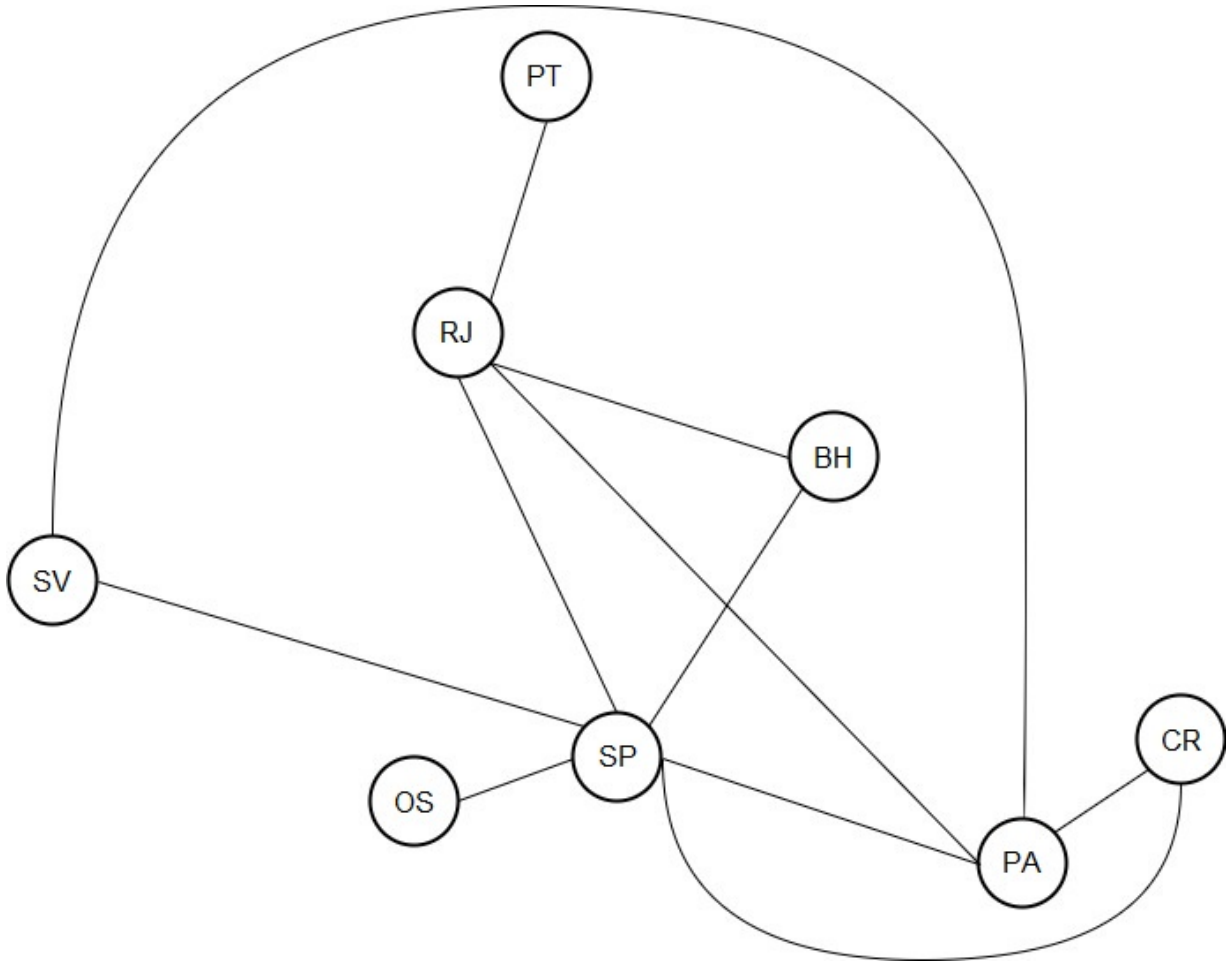


Figura 2.4: Grafo G oriundo da relação entre cidades.

E, pronto! Conseguimos modelar nosso domínio em um grafo. Isso por si só já representa um avanço considerável, porque tivemos a capacidade de abstrair o problema debatido em um nível que nos possibilitou entender quais seriam os vértices e arestas do grafo em questão. Contudo, uma lacuna ainda não foi preenchida. Como

representar o grafo G em uma estrutura computacional que nos possibilite realizar operações sob ele?

Lembre-se, você é um analista de sistemas, e não é somente qualquer um, é um dos bons. Tem um amplo conhecimento sobre desenvolvimento de software e Teoria dos Grafos e quer pôr o melhor dos dois mundos em uma coisa só. Por isso, você sente essa necessidade de encher esse espaço vazio. Por meio de uma linguagem de programação essa tarefa se torna possível.

Qual linguagem escolher? Por ser uma linguagem que oferece um sólido suporte a Orientação a Objetos, escolho a linguagem de programação Java para construção do nosso projeto durante todo o livro. Mas nada impede que você implemente as soluções propostas aqui e suas próprias em uma linguagem de seu gosto em particular. Inclusive, a utilização de uma outra linguagem traz o poder da comparação. Quem sabe você não consegue fazer melhor do que foi feito aqui?

A versão do compilador (JDK) e as ferramentas de desenvolvimento também são de escolhas suas. Nada impede que você utilize *IDEs*, como o *Eclipse* ou o *Netbeans*. Em relação à versão da JDK, a escolha é livre pois o nosso foco não é o estudo de características do Java, mas, sim, o estudo da Teoria os Grafos usando o Java como uma forma de implementar a teoria aprendida. Não utilizaremos recursos específicos de uma determinada versão, portanto, desde que você utilize uma JDK 5 ou superior, não teremos problemas.

O projeto está estruturado de uma forma bem simples, como a imagem mostra:

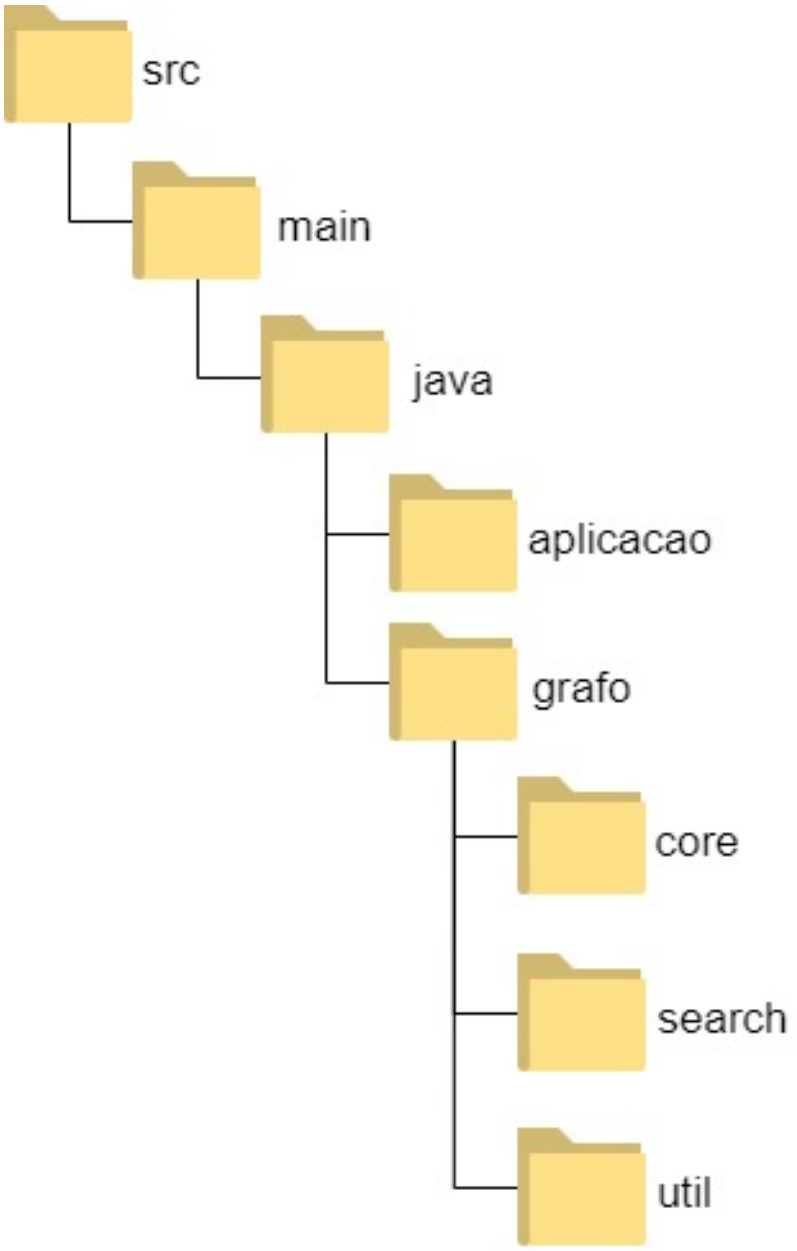


Figura 2.5: A estrutura do projeto.

As pastas `src`, `main` e `java` seguem o padrão **Maven** de empacotamento amplamente conhecido e difundido no mundo do desenvolvimento de software. Abaixo do pacote `java` está o que nos importa, ali começa a estrutura do projeto. Neste nível temos duas pastas, `aplicacao` e `grafo`, as quais representam, respectivamente, a camada de aplicação e a camada de domínio do projeto. A primeira é responsável por oferecer uma via de acesso ao sistema

para o usuário. Nela, criaremos todos os artifícios necessários para que a usabilidade do usuário seja a melhor possível. Já a segunda é responsável pelo tratamento de todo o domínio da aplicação, ou seja, será esta camada que entenderá tudo sobre grafos e as aplicações que desenvolveremos, e que proverá formas para que a camada de aplicação possa usufruir das funcionalidades.

Voltando nossa atenção para a segunda camada, dentro deste pacote temos as seguintes pastas: `core`, `search` e `util`. Respectivamente, elas vão concentrar todo o conceito sobre grafos e suas estruturas, o cerne; o conjunto de ferramentas necessárias para realizar buscas; e componentes utilitários que podem ser usados esporadicamente. Não se preocupe neste momento com detalhes técnicos, veremos questões como essas mais adiante.

Neste capítulo, vamos construir as fundações que suportarão nossa aplicação, e vamos definir como será um grafo e suas estruturas auxiliares seguindo o paradigma orientado a objetos. Logo, tudo criado aqui será dentro pacote `core`, pois este é o núcleo da nossa aplicação.

No paradigma orientado a objetos, quando queremos representar algo utilizamos uma classe. Este "algo" pode ter `n` formas, pode ser um conceito, uma operação, um elemento do seu problema etc. Não tente decorar as múltiplas formas que podem originar uma classe, é tempo perdido. Ao contrário, tenha em mente que o que vai realmente definir uma classe é o domínio da sua aplicação, ou seja, o problema que você está tratando (Evans, Eric; 2017). No nosso caso, um grafo se torna uma classe porque ele representa um conceito, visto anteriormente, que possui várias características e operações, e para o contexto da nossa aplicação faz sentido que ele tenha essa forma.

Portanto, dentro do pacote `core` crie a classe `Grafo`.

```
src > main > java > grafo > core > Grafo.java
```

```
package main.java.grafo.core;
```

```
public class Grafo {

}
```

Neste momento, nossa classe não representa nada porque ainda está vazia. Para que ela comece a representar algo e possa, consequentemente, ser uma classe de fato, precisamos identificar quais são suas características e o que ela pode fazer. Sabemos que grafos têm vértices. Tal fato representa uma característica de um grafo. Todo grafo tem vértices, logo, precisamos fazer com que nossa classe represente isso.

```
src > main > java > grafo > core > Grafo.java

package main.java.grafo.core;

public class Grafo {
    private List<Vertice> vertices = new ArrayList<Vertice>();
}
```

Pronto! Adicionamos esta característica e o nosso grafo acaba de ganhar vértices. Entretanto, para nossa aplicação, o que é um vértice? Não construímos esta definição ainda. Será que para o paradigma orientado a objetos um vértice poderia se tornar uma classe? Pensemos.

Voltando algumas páginas e dando uma olhadinha no grafo que montamos, vimos que cada vértice tinha uma espécie de rótulo que ajudava em sua identificação e, voltando mais um pouquinho, expusemos que assim como o rótulo ele também possui um grau. Então, podemos afirmar que para o nosso domínio um vértice deve possuir um rótulo e um valor de grau. Portanto, no mesmo pacote da classe `Grafo`, crie a classe `Vertice`.

```
src > main > java > grafo > core > Vertice.java

package main.java.grafo.core;
```

```
public class Vertice {
    private String rotulo;
    private int grau;

    public String getRotulo(){
        return this.rotulo;
    }
}
```

Finalmente! Temos uma representação sólida de um grafo que possui vértices, os quais por sua vez possuem rótulos e graus. Mas faz sentido existir um vértice sem rótulo?

Para o nosso domínio isso não tem cabimento porque cada filial tem uma identificação baseada na sua localidade. Logo, precisamos forçar essa relação de alguma forma. Precisamos deixar claro que a criação de um vértice só faz sentido se ele possuir um rótulo. Sendo assim, adicione um método construtor personalizado na classe `Vertice`.

`src > main > java > grafo > core > Vertice.java`

```
package main.java.grafo.core;

public class Vertice {
    private String rotulo;
    private int grau;

    public Vertice(String rotulo) throws Exception {
        boolean isRotuloNullOrBlank = rotulo == null || rotulo != null &&
"".equals(rotulo.trim());
        if (isRotuloNullOrBlank) {
            throw new Exception("Não é permitida a inclusão de vértices
com rótulo em branco.");
        }
        this.rotulo = rotulo;
    }

    public String getRotulo(){
        return this.rotulo;
```

```
    }
}
```

Mas e se o rótulo passado para o vértice estiver em branco? Como, conceitualmente, nosso domínio se comporta perante isso? Assim como para casos de ausência de rótulo, casos como este devem ser tratados da mesma forma. Um vértice não pode aceitar que seu rótulo seja nulo ou em branco. Por esse motivo, no método construtor existe a verificação que só aceita rótulos que cumpram estes requisitos básicos. Caso um rótulo não cumpra, nossa classe emite um aviso de alerta.

Este aviso de alerta, dentro do paradigma orientado a objetos, é uma *exception*. Avisos como este sinalizam que alguma regra do nosso domínio foi quebrada e que o sistema deve responder à altura. A nossa carregará uma mensagem informativa.

Voltando para a classe `Grafo`, será que além dos vértices ela não possui mais nenhuma característica? Mais uma vez, pensemos. Quantos vértices um grafo deve aceitar? A resposta para essa pergunta reside com os responsáveis pela rede de envio e recebimento de documentos da empresa. Então, prontamente você mandou um e-mail pedindo essa informação à área responsável e eles responderam que atualmente existem somente 8 filiais junto com a sede por todo o país, mas talvez em um futuro próximo a empresa pode sofrer uma expansão agressiva, logo, não é possível aferir com certeza uma quantidade.

Portanto, através dessa informação corporativa podemos concluir que o nosso grafo deve aceitar qualquer quantidade de vértices, desde que essa quantidade seja maior ou igual a 1. E assim, mais uma vez precisamos deixar explícito que a criação de um grafo só faz sentido se essa regra for respeitada. Adicione, então, dois métodos construtores à classe `Grafo`.

`src > main > java > grafo > core > Grafo.java`

```
package main.java.grafo.core;
```

```
public class Grafo {
    private List<Vertice> vertices = new ArrayList<Vertice>();

    public Grafo() {
        qtdMaximaVertices = 10;
    }

    public Grafo(int qtdVertices) {
        if (qtdVertices <= 0) {
            throw new IllegalArgumentException("A quantidade máxima de
vértices deve ser maior ou igual à 1");
        }
        qtdMaximaVertices = qtdVertices;
        isQtdMaximaDefinida = true;
    }
}
```

Antes de mais nada, por que dois construtores? A criação de um bom projeto de software é algo que envolve intenso conhecimento do domínio e uma flexibilidade sutil. A adição de dois construtores a essa classe une essas duas características, pois nem sempre a quantidade de vértices é previamente conhecida e isso é atingido através da parametrização da quantidade de vértices.

No primeiro construtor - sem quantidade de vértices definida - a quantidade máxima de vértices é definida para 10. Já no segundo, é verificado se a quantidade passada respeita a regra mencionada acima. Caso não respeite, um alerta, oferecido pela própria linguagem Java, é emitido. Uma vez que a regra foi respeitada, a quantidade é armazenada no atributo `qtdMaximaVertices` e o atributo *flag* `isQtdMaximaDefinida` é acionado.

Falamos aqui de vários atributos da nossa classe, mas não declaramos nenhum deles. Então, sua classe `Grafo` deve ter essa cara.

`src > main > java > grafo > core > Grafo.java`

```java
package main.java.grafo.core;

public class Grafo {

    private int qtdMaximaVertices;
    private boolean isQtdMaximaDefinida;
    private int qtdAtualVertices = 0;
    private Map<String, Integer> rotulosEmIndices = new HashMap<String,
Integer>();
    private List<Vertice> vertices = new ArrayList<Vertice>();

    public Grafo() {
        qtdMaximaVertices = 10;
    }

    public Grafo(int qtdVertices) {
        if (qtdVertices <= 0) {
            throw new IllegalArgumentException("A quantidade máxima de
vértices deve ser maior ou igual à 1");
        }
        qtdMaximaVertices = qtdVertices;
        isQtdMaximaDefinida = true;
    }
}
```

Os atributos `qtdAtualVertices`, `rotulosEmIndices` e `vertices` são inicializados em tempo de declaração, pois seus valores não mudam independentemente da forma como um grafo for criado.

Um fator que pode chamar a atenção é a escolha feita para o tipo do atributo `rotulosEmIndices`. O tipo `Map`, assim como sua implementação, o `HashMap`, são tipos que representam dicionários. Dicionários são listas, cujas posições são representadas por identificadores em vez de índices numéricos, assim como um dicionário de verdade, em que cada palavra (identificador) está ligado a um significado. Através de um identificador é possível ter acesso **O(1)** a um valor, em outras palavras, um acesso direto sem a necessidade de percorrer a lista inteira verificando elemento pós elemento.

No nosso caso, o dicionário de rótulos em índices possuirá como identificador um rótulo de um vértice e, como valor atribuído a este identificador, a posição numérica que aquele vértice possui na lista de vértices. Desta forma, viabilizamos um acesso **O(1)** de um objeto do tipo `Vertice` a partir de seu rótulo. A escolha por este tipo de design fará mais sentido no capítulo 3.

Até agora só falamos sobre as características de um grafo, mas e as ações dele? O que ele poderá fazer? O usuário, neste momento, certamente terá a vontade de adicionar a sede da empresa e suas filiais ao grafo e depois obter algum tipo de relatório disso. Em vista disso, devemos concluir que, para o nosso domínio, um grafo pode adicionar e recuperar seus vértices, correto? Então, adicione à classe `Grafo` os métodos `adicionarVertice(rotulo)` e `getVertices()`.

`src > main > java > grafo > core > Grafo.java`

```
public void adicionarVertice(String rotulo) throws Exception {
    if (qtdAtualVertices <= qtdMaximaVertices - 1) {
        Vertice novoVertice = new Vertice(rotulo);
        this.vertices.add(novoVertice);
        this.rotulosEmIndices.put(rotulo, qtdAtualVertices);
        qtdAtualVertices++;
    } else {
        throw new Exception("A quantidade de vértices permitida (" +
qtdMaximaVertices + ") foi excedida.");
    }
}

public List<Vertice> getVertices(){
    return this.vertices;
}
```

O método `adicionarVertice(rotulo)`, ao executar sua função, verifica se a quantidade máxima de vértices foi atingida e, caso não tenha sido, adiciona o novo vértice ao grafo com o rótulo passado como parâmetro. Caso a quantidade máxima tenha sido extrapolada, um alerta é emitido. O método `getVertices()` retorna os vértices do grafo

para que a camada de aplicação possa manipulá-los da forma que achar melhor.

Com isto, finalizamos todo o nosso trabalho na camada de domínio. Então, o que resta é mostrar como usar.

```
Grafo grafo = new Grafo();

grafo.adicionarVertice("RJ");
grafo.adicionarVertice("SP");
grafo.adicionarVertice("BH");
grafo.adicionarVertice("PT");
grafo.adicionarVertice("OS");
grafo.adicionarVertice("SV");
grafo.adicionarVertice("CR");
grafo.adicionarVertice("PA");

System.out.println("O grafo G possui os seguintes vértices:");
System.out.println();
for(Vertice vertice : grafo.getVertices()) {
    System.out.println("- Vértice " + vertice.getRotulo());
}
```

Essa amostra de código nos mostra como criar um grafo utilizando a quantidade padrão de vértices, adicionar vértices e imprimi-los. É possível executar este trecho de código em qualquer classe dentro de um método `main`. Se você fizer este teste, espere o seguinte resultado.

```
O grafo G possui os seguintes vértices:

- Vértice RJ
- Vértice SP
- Vértice BH
- Vértice PT
- Vértice OS
- Vértice SV
- Vértice CR
- Vértice PA
```

Com isso, fechamos uma primeira versão para teste da nossa aplicação. Acho que a galera da área de documentos vai adorar. Caso você esteja utilizando alguma ferramenta de desenvolvimento (IDE) para o projeto, será muito fácil executá-lo e fazer o *debugging*, mas caso esteja utilizando o editor de texto do sistema operacional, compilar e executar serão tarefas manuais.

Todo o sistema operacional possui um *prompt* de comando ou um *console*. Uma vez que você esteja com ele aberto, navegue até pasta `src` do seu projeto. Lá, execute o comando `javac -classpath . caminho_da_sua_classe.java` para todas as classes Java que existam. Ele compilará sua classe e as dependências dela, e gerará o arquivo *bytecode* com a extensão *class*. Caso o seu programa tenha erros de escrita ou avisos ignorados, o compilador vai avisar.

Uma vez que você compilou todos os arquivos `.java` que criamos neste capítulo, está na hora de pôr para funcionar. Também na pasta `src`, execute o comando `java caminho.classe.com.método.main`, por exemplo: `java main.java.aplicacao.Aplicacao`. Note que não há extensão de arquivo nessa instrução. É assim mesmo, não se desespere. A previsão é que o seu programa funcione sem problemas.

Está sentindo falta de algo? Sim, algo está faltando. Ainda não descobriu o que é? **Onde estão as arestas? E por que não implementamos o grau de um vértice?**

Até agora não implementamos nada sobre arestas na nossa aplicação por um motivo. Para podermos adicioná-las e formar um grafo por completo, precisamos antes aprender como representar as conexões entre vértices e arestas. Este ponto é crucial para que possamos seguir em frente e dar vida às nossas arestas. A chave para este segredo está no próximo capítulo.

2.3 Conclusão

Entender conceitualmente o que é um grafo não é uma tarefa difícil, mas é necessário ter uma mente aberta, pois o que vem depois disso são conceitos complementares e características que podem tornar a visão do todo um pouco nebulosa. Logo, quando olhamos o grande cenário a primeira impressão que temos é de ser complexo, mas de passo em passo isso pode ser desmistificado.

Assim como o conceito matemático pode ser amedrontador, a transformação desse aglomerado de conhecimento para o campo computacional pode ser algo que cause bastante confusão, mas o segredo é transformar todo o passo a passo teórico conhecido em um procedimento organizado e prático, utilizando como apoio estruturas de dados e paradigmas de programação.

Resumo

- Um grafo G(V,E) é uma estrutura formada por dois conjuntos: V e E.
- O conjunto V representa o conjunto dos vértices de um grafo.
- O conjunto E representa o conjunto das arestas de um grafo.
- Um caminho é uma sequência de vértices e arestas organizados de forma alternada.
- Um caminho simples é todo aquele em que nenhum vértice, e consequentemente nenhuma aresta, se repete.
- Um circuito é todo aquele caminho em que tanto o vértice inicial quanto o final são os mesmos.
- Um circuito simples é todo aquele circuito em que nenhum vértice se repete, exceto o inicial e o final.
- O grau de um vértice é definido pela quantidade de arestas conectadas a ele.

- Grafos orientados são similares em estruturas e propriedades a grafos não orientados, exceto que suas arestas possuem direções.
- A direção muda a análise sob os caminhos de um grafo, pois se antes era possível ir de um vértice a outro, desde que estivessem conectados, agora talvez não seja mais.

Capítulo 3
Para onde foram as arestas?

Terminamos o capítulo anterior com duas perguntas ainda não respondidas. Para onde foram as arestas das quais tanto falamos no nosso projeto? E por que não implementamos a funcionalidade que calcula o grau de um vértice? Para responder a essas e outras perguntas precisamos antes entender um conceito primordial, que abrirá uma nova porta nesta aventura e expandirá os nossos limites além do alcance.

3.1 Matriz de adjacência e matriz de incidência

Aqui jaz a última lição vital que abrirá as portas para as diversas aplicações que um grafo pode ter. Parabéns, você encontrou a toca do coelho.

Adjacências e incidências são conceitos que sempre andam de mãos dadas, sendo o primeiro relacionado aos vértices, e o segundo, às arestas. A partir do momento em que encontramos a adjacência entre dois vértices também encontramos a incidência da aresta que os liga. Talvez essa relação não seja tão visível em grafos não orientados quanto é em dígrafos, mas em ambos os contextos elas existem.

Para que possamos entender o que é uma matriz de adjacência ou incidência, precisamos antes entender o que é uma matriz. Quando digo matriz você provavelmente se lembra dos tempos de escola, nas aulas de matemática, onde viu esse assunto pela primeira vez e achou que era algo de outro mundo. Sim, é dessa matriz que estamos falando também. Também? Existe mais de uma? Sim, existe. Estamos nos referindo aqui a dois tipos de matrizes, a

primeira é a matemática, que já conhecemos, e a segunda é a computacional.

Na matemática, uma matriz se parece com uma tabelinha cheia de valores, chamados de termos, e que possui m linhas por n colunas, onde todos os valores dessas linhas e colunas são sobre um determinado conjunto (Callioli, Carlos; Domingues, Hygino; Costa, Roberto; 1978). Já uma matriz computacional é uma forma de representação da matriz matemática oferecida pelas linguagens de programação, para que possamos manipular dados que possuam duas dimensões.

Um pouco abstrato, não é? Fica mais fácil quando pensamos em um jogo de batalha naval, daqueles de tabuleiro. Geralmente, o tabuleiro, feito de plástico ou papelão, se parecia com uma tabela que, por sua vez, possuía letras representando as linhas e números, as colunas. Sem um jogador ver o jogo do outro, posicionavam-se os navios de guerra utilizando essas coordenadas e o jogo começava. Portanto, nesse caso, o tabuleiro seria a matriz, as letras e números seriam as duas dimensões, e os dados manipulados seriam as posições dos navios. Através da combinação linha com coluna (latitude com longitude) é possível afirmar a posição exata de um barco ou água (para o seu azar).

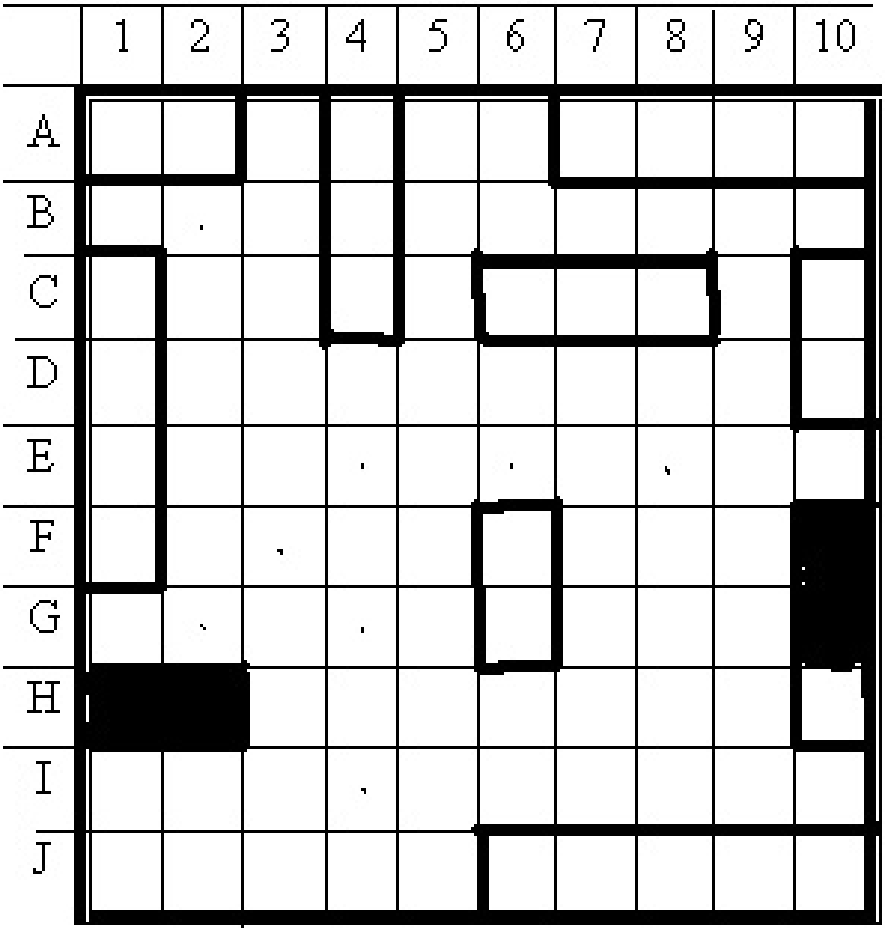


Figura 3.1: Batalha naval. Fonte: https://pt.wikipedia.org/wiki/Batalha_naval_(jogo)#/media/File:Schiffeversenk.jpg

Computacionalmente a lógica é a mesma, você cria a matriz, informa suas dimensões e manipula seus dados através da mesma combinação, linha com coluna. Dimensões de uma matriz expressadas em uma linguagem de programação como a que escolhemos começam a partir do número 0, ou seja, se informamos que nossa matriz terá cinco linhas deveremos contar começando do número 0, como primeira linha, e indo até o número 4 como quinta linha.

Matriz de adjacência

Agora que já sabemos o que é uma matriz, precisamos entender o que é a adjacência entre dois vértices.

DEFINIÇÃO:

É dito adjacente todo vértice que é ligado a outro por uma aresta.

O conceito de adjacência sempre envolve o próximo vértice ligado por uma aresta independentemente da maneira com que o grafo possa ser desenhado, então, o próximo pode estar à frente, atrás, ao lado etc. Para dois vértices `v0` e `v1`, ambos ligados por uma aresta sem direção, podemos afirmar que `v1` é adjacente a `v0`, se olharmos pelo ponto de vista de `v0`, e `v0` é adjacente a `v1` olhando por `v1`.

Computacionalmente, isso resulta em uma matriz M quadrada `nxn`, onde `n` simboliza o número de vértices do grafo. O termo "quadrado" existe para dar a ideia de que o número de linhas é igual ao de colunas, assim como um quadrado, que possui seus lados iguais. Portanto, se um grafo G possui 5 vértices, sua matriz terá 5 linhas por 5 colunas com 25 células no total.

Lembra da combinação linha com coluna do jogo "Guerra Naval" que vimos parágrafos antes? Aqui ela existe na forma de M_{ij}, onde `i` representa a linha e `j` a coluna da matriz, portanto, para M_{01} estaremos nos referindo à primeira linha com a segunda coluna da matriz. Sempre que houver o número 1 em uma célula é porque existe ali uma aresta que conecta um vértice a outro, representados respectivamente pela linha e coluna da combinação. E assim como no jogo "Guerra Naval", você pode acabar atirando na água. É o caso do número 0 em uma célula, que representa que não há conexão entre vértices (Even, Shimon; 1979).

Acordo:

Vamos sempre considerar `i` como sendo o vértice inicial, e `j`, como o vértice final da conexão entre dois vértices, ou seja, para uma aresta `e` que conecte os vértices `u` e `v`, então `i` será `u` e `j` será `v`.

Para grafos não orientados - aqueles cujas arestas não possuem direção - a matriz de adjacência passa por um fenômeno de simetria: um lado refletindo o outro. Como a adjacência de um vértice é bilateral neste caso, para todo $M_{ij}=1$ é necessário realizar também $M_{ji}=1$. Em outras palavras, se `v0` está ligado a `v1` então `v1` também está ligado a `v0`. Porém, para casos de grafos orientados - aqueles em que as arestas possuem direção - a simetria não acontece de forma espontânea e qualquer configuração de matriz se torna possível pois, como já visto no capítulo 2, direções mudam tudo (Even, Shimon; 1979). Em ambos os casos, a diagonal da matriz pode permanecer repleta de zeros caso não haja arestas recursivas, mas caso haja é possível representá-las também.

Definição:

Seja uma matriz M quadrada `nxn`, onde `n` representa o número de vértices do grafo, considerada a matriz de adjacência se ela consegue mapear para cada vértice todos os seus vértices adjacentes em suas correspondentes coordenadas na matriz.

Aplicando os conceitos vistos, a matriz de adjacência do grafo a seguir será:

	v0	v1	v2	v3	v4
v0	0	0	0	1	0
v1	0	0	0	1	1
v2	0	0	0	1	0
v3	1	1	1	0	1
v4	0	1	0	1	0

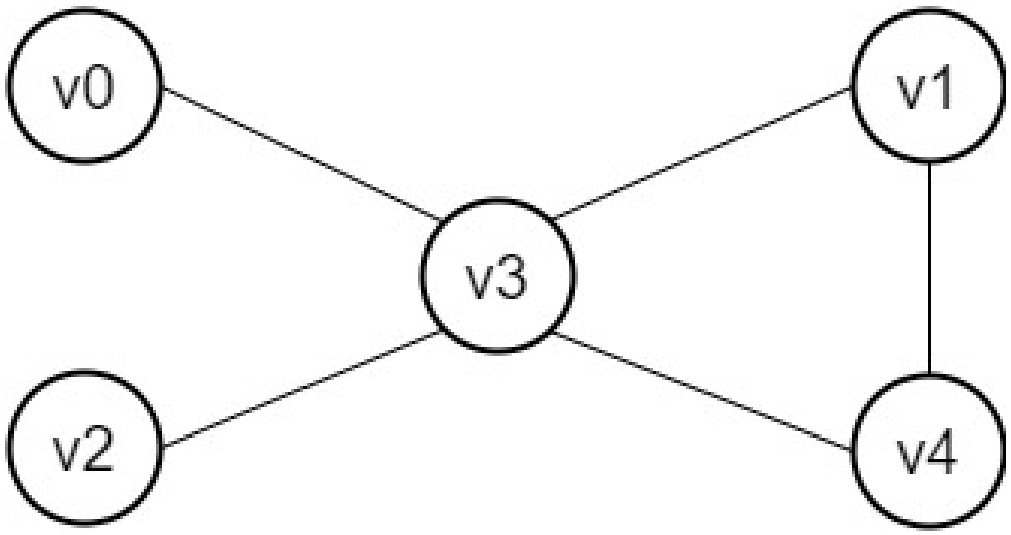


Figura 3.2: Um grafo e sua matriz de adjacência

E se um grafo possuísse arestas paralelas, como isso poderia ser representado? Mais uma vez precisamos recorrer ao conceito antes de solucionar o problema.

Definição:

Arestas são consideradas paralelas se, e somente se, possuírem os mesmos vértices iniciais e finais. Esta regra vale tanto para grafos não orientados quanto em dígrafos (Even, Shimon; 1979).

Voltando à pergunta, esta questão é contornada adicionando mais um valor à matriz. Sempre que entre dois vértices existirem arestas paralelas, para seu respectivo M_{ij}, em vez de 1 adicionaremos 2 (Bondy, J. A.; Murty, U. S. R.; 1976). Obviamente, o novo valor é justificado neste caso porque são duas arestas adjacentes a um vértice, e não mais uma.

Tocar neste assunto para um grafo não orientado é algo um pouco estranho, porque você pode pensar: "Para que dois caminhos, já que a aresta não possui uma direção?". É, realmente devo dar o braço a torcer, em um grafo não orientado, arestas paralelas não são lá tão triviais assim, mas podem existir. Como vimos no capítulo 1, o que define a necessidade da existência de um vértice ou uma aresta é o problema tratado, vide o problema das Sete Pontes de Königsberg. Contudo, para grafos orientados, o conceito de arestas paralelas fará todo sentido, pois uma vez que existe uma direção demarcando o sentido do trajeto somente será possível adicionar um caminho de volta através de uma aresta paralela com sentido contrário.

Matriz de incidência

Como dito anteriormente, a incidência tem seus olhos voltados para as arestas de um grafo. Tal fato tem consequências que se refletem nas dimensões de sua matriz. Mas o que é uma incidência?

DEFINIÇÃO:

É incidente toda aresta que vai em direção a um vértice.

Estranha essa definição, não? Calma, é normal que você estranhe um pouco porque em grafos não orientados é impossível determinar a direção de uma aresta. Logo, neste caso, uma aresta é incidente aos dois vértices que ela conecta dependendo sempre do ponto de vista. Já em grafos orientados ou dígrafos, a história é outra. Como neste contexto temos uma direção explícita, é possível determinar onde uma aresta nasce (*start vertex*) e para onde ela vai (*end vertex*), ou seja, onde ela incide.

Lembra das consequências nas dimensões da matriz no parágrafo anterior? Então, como a incidência é uma relação que considera primariamente as arestas e, secundariamente os vértices, isso resulta em uma matriz M $_{v \times e}$ (v por e), onde v representa os todos os vértices, e e, todas as arestas do grafo. Para qualquer combinação linha com coluna, M_{ij}, onde i representa uma linha e j uma coluna, pode haver os valores 0, 1 ou 2, que representam o número de vezes que uma aresta é incidente a um vértice (Bondy, J. A.; Murty, U. S. R.; 1976).

DEFINIÇÃO:

Seja uma matriz M $_{v \times e}$ (v por e), onde v representa os vértices, e e, as arestas do grafo, ela é considerada a matriz de incidência se ela consegue mapear para cada aresta todos os seus vértices incidentes em suas correspondentes coordenadas na matriz.

Assim como na matriz de adjacência, nesta aqui também é possível mapear todas os tipos de arestas que um grafo pode ter e o acordo estabelecido lá também vale aqui. Contudo existe uma exceção à

regra relacionada aos valores que a matriz pode assumir. Somente haverá o valor dois em casos em que houver duas arestas incidentes ao mesmo vértice. Mas até aí nada mais que o óbvio, não?! Porém, a diferença é que isso acontecerá somente em casos de arestas recursivas em vez de arestas paralelas, como acontece nas matrizes de adjacência. Isso se deve ao fato de a recursividade ser o único elemento que incide duas vezes ao mesmo vértice.

Com a imagem adiante ficará tudo mais claro. As arestas foram nomeadas para que ficasse mais simples identificar qual aresta incide em qual vértice.

	e1	e2	e3	e4	e5	e6	e7	e8
v0	1	0	0	0	0	1	0	0
v1	0	0	0	1	1	0	0	0
v2	0	1	0	0	0	0	2	1
v3	1	1	1	1	0	0	0	1
v4	0	0	1	0	1	1	0	0

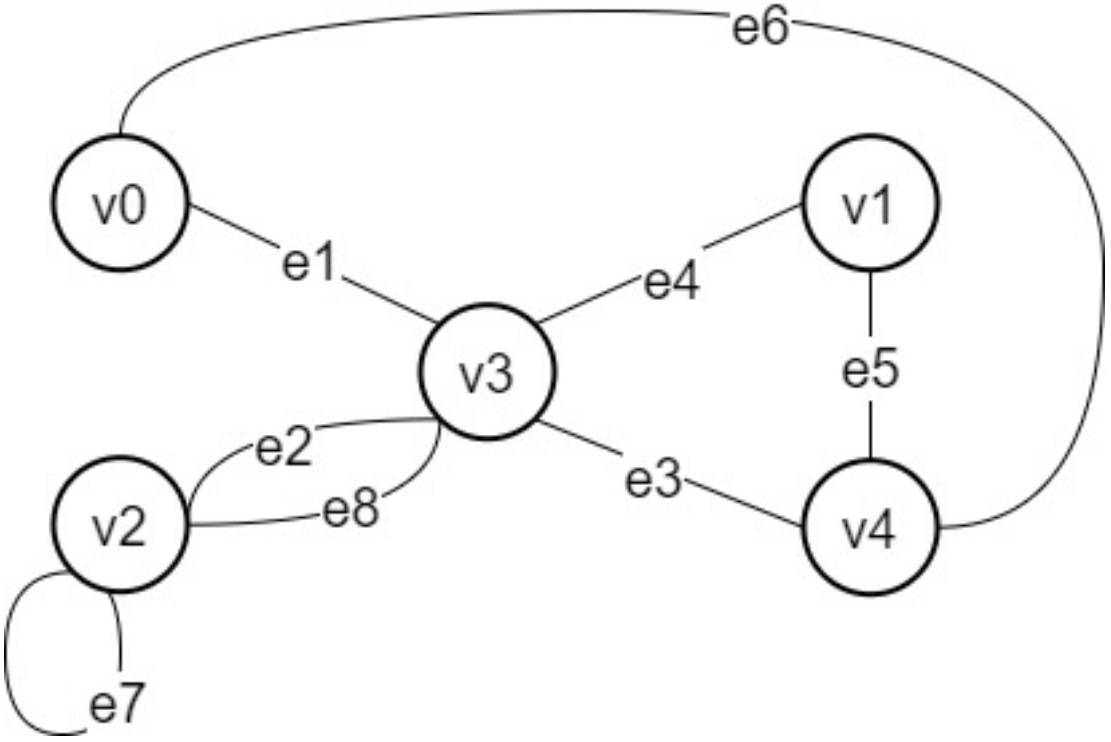


Figura 3.3: Um grafo e sua matriz de incidência.

É sempre bom ter o maior conhecimento possível sobre as formas de representar as relações entre vértices e arestas nos diferentes

tipos de matriz porque, como veremos à frente, isso influencia em funcionalidades que um grafo pode oferecer, como o seu grau.

O grau de um vértice

Como já passamos por esse assunto no capítulo 2, acho que somente vale a pena relembrar o que é o grau de um vértice, para que em seguida possamos entender alguns novos conceitos.

> **DEFINIÇÃO:**
>
> O grau de um vértice em um grafo consiste no número de arestas que são incidentes a ele, e é representado pela função `d(v)` ou `deg(v)` (Harary, Frank; 1970).

Antes de aprendermos como calcular o grau utilizando uma das matrizes já vistas, vamos com um pouco mais de teoria.

> **CURIOSIDADE**
>
> O nome da função `deg(v)` se dá pelo fato de que a palavra *degree* em inglês significa grau.

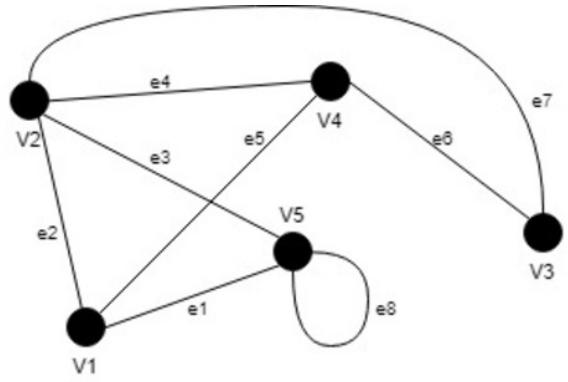
Dado um grafo G não orientado com *x* vértices e *y* arestas, como calcular o somatório de graus de seus vértices? Simples! O somatório de graus de qualquer grafo, nas mesmas condições, se dá pelo dobro do número de arestas (Bondy, J. A.; Murty, U. S. R.; 1976).

$$\sum_{v \in V} d(v) = 2 \text{ . quantidade de arestas}$$

Figura 3.4: Teorema do somatório de arestas

Por que é necessário a multiplicar por dois? Se uma aresta sempre incide em dois pontos, sendo recursiva ou não, então ela contribui em dois para o somatório, logo a multiplicação. Inclusive este foi o primeiro teorema proposto por Euler (Harary, Frank; 1970).

No primeiro grafo do capítulo 2 tínhamos cinco vértices e oito arestas. Aplicando essa fórmula, o somatório é igual a 16. Mas vamos detalhar um pouco mais para ver se está tudo certo?



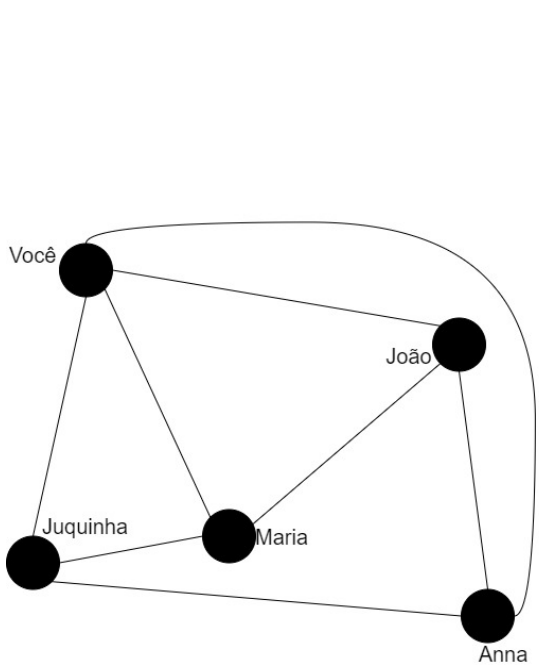
Vértice	d(v)
v1	3
v2	4
v3	2
v4	3
v5	4

3 + 4 + 2 + 3 + 4 = 16

Figura 3.5: Aplicação do teorema de somatório de graus

Entretanto, uma questão ainda não foi resolvida: como calcular o grau de um vértice usando a matriz de adjacência? Já sabemos como calcular o somatório total de graus e o que é o grau de um vértice, mas ainda não como calculá-lo. Imagine um grafo gigantesco, como um que represente seus amigos em sua rede social favorita, assim como a imagem do meu LinkedIn Maps do capítulo 1. Você quer saber quantos amigos, incluindo você, tem aquele seu antigo amigo de escola. Como resolver?

O primeiro passo a ser tomado é montar a matriz de adjacência e verificar qual foi o acordo feito para as linhas e colunas. Quando digo "acordo", refiro-me ao que fizemos parágrafos antes, onde definimos que as linhas da matriz seriam os vértices iniciais de uma conexão entre vértices, e as colunas, os finais. Geralmente essa é a convenção usada. Feito isso, determine qual é o seu tipo de grafo, orientado ou não orientado. Após, escolha seu amigo e faça a contagem de arestas incidentes a ele. Isso pode ser feito contando a quantidade de células diferentes de 0 em seu eixo incidente. *Voilà*, o grau do vértice está calculado (Bondy, J. A.; Murty, U. S. R.; 1976).



Eixo incidente de João

	Você	João	Maria	Juquinha	Anna
Você	0	1	1	1	1
João	1	0	1	0	1
Maria	1	1	0	1	0
Juquinha	1	0	1	0	1
Anna	1	1	0	1	0

Figura 3.6: Calculando a quantidade amigos de João.

Determinar qual é o tipo de grafo e verificar qual convenção utilizada são fatores importantes, pois tais determinam qual será o eixo incidente da matriz. No nosso caso estaremos sempre falando de colunas. Por tais motivos, os respectivos valores `d(v)` para a imagem da aplicação do teorema são justificados.

3.2 E as arestas chegam ao projeto

No capítulo anterior, criamos toda a estrutura do nosso projeto e inclusive iniciamos o desenvolvimento de algumas das suas funcionalidades. Utilizando todo esse ambiente pronto, vamos colocar em prática tudo o que foi aprendido, visando obviamente inserir mais funcionalidades na aplicação.

Nosso primeiro objetivo deve ser definir um local apropriado para posicionar nossa matriz, até porque ela será o foco do nosso trabalho. Uma vez posicionada corretamente, devemos pensar em como adicionaremos valores nela, de forma que corresponda à realidade do grafo, ou seja, as conexões entre seus vértices. E, finalmente, construir meios de acesso para as possíveis operações de que o seu projeto precisa.

Para atingir nossa primeira meta será necessário pensar em uma estrutura que possa concentrar toda a responsabilidade relacionada à matriz. Como estamos desenvolvendo sob o paradigma orientado a objetos, o que seria melhor que uma classe para resolver esse problema? Crie a classe `MatrizAdjacencia` no pacote `core`, o mesmo que usamos no capítulo 2 para a criação da classe `Grafo`.

Agora sim, é disso que precisamos! Nela poderemos criar a matriz de fato e desenvolver tudo o que for necessário. Mas que forma essa matriz terá dentro da classe? Mais uma vez será necessário recorrer à orientação a objetos.

Classes agem como modelos de abstração que conseguem concentrar características e ações debaixo do mesmo teto. Uma característica que nossa classe `MatrizAdjacencia` tem é a matriz em si. Logo, crie um atributo que a representa e dê um belo nome para manter seu código limpo, como `matriz` ou `matrizDeAdjacencia`. Este será o principal atributo da classe, mas como todo herói precisa de um escudeiro, aqui não será diferente. A matriz precisa dos vértices de seu grafo para recuperar mais detalhes caso seja necessário. Então, até agora o que temos é o seguinte:

```
src > main > java > grafo > core > MatrizAdjacencia.java

package main.java.grafo.core;

public class MatrizAdjacencia{
    private int[][] matriz;
    private List<Vertice> vertices;
}
```

Até o momento, tudo bem, certo? Contudo, olhar para essa classe me deixa com uma certa dúvida. Como essa lista conterá os vértices que já adicionamos no grafo? Como esses valores chegarão aqui? Ele precisa ser passado para a classe em algum ponto, mas que lugar seria o ideal para isso acontecer?

Temos de concordar que uma matriz de adjacência só deve existir se existirem vértices no grafo. Logo, o que precisamos aqui é criar um construtor para essa classe que possua um parâmetro que represente essa necessidade. Desta forma tornamos explícita essa relação de existência: só será possível criar uma matriz de adjacência se os vértices do grafo forem passados para ela. Assim, não damos chances para erros semânticos.

```
src > main > java > grafo > core > MatrizAdjacencia.java

package main.java.grafo.core;

class MatrizAdjacencia{
    private int[][] matriz;
    private List<Vertice> vertices;
    private int qtdVertices;

    public MatrizAdjacencia(List<Vertice> vertices){
        this.vertices = vertices;
        this.qtdVertices = vertices.size();
        matriz = new int[qtdVertices][qtdVertices];
        inicializarMatriz();
    }
}
```

Uma vez que temos os vértices em mãos é possível determinar quantos existem no grafo. E, uma vez que sabemos dessa quantidade, já podemos definir as dimensões da nossa matriz e o valor do atributo `qtdVertices`. Um outro ponto que você deve ter notado é a chamada à função `inicializarMatriz()`. É necessário inicializar a matriz, mas com o que e por quê? Em sua forma primordial, um grafo não possui arestas, a menos que você as adicione, e isso deve ter alguma representação. Em seções anteriores já abordamos qual é a forma exata a que este fenômeno adere, o número 0. Logo, inicialmente devemos adicionar 0's em nossa matriz para representar tal ocasião, deixando-a pronta para a adição de arestas no grafo.

`src > main > java > grafo > core > MatrizAdjacencia.java`

```
private void inicializarMatriz(){
    for(int i=0; i<matriz.length; i++){
        for(int j=0; j<matriz[i].length; j++){
            matriz[i][j] = 0;
        }
    }
}
```

Pronto, a matriz já foi criada e está pronta para ser usada. O que nos resta agora é prover as funcionalidades necessárias para que possamos adicionar e recuperar as arestas com facilidade.

`src > main > java > grafo > core > MatrizAdjacencia.java`

```
public void adicionarAresta(int indiceVerticeInicial, int
indiceVerticeFinal) {
    Vertice verticeInicial = vertices.get(indiceVerticeInicial);
    Vertice verticeFinal = vertices.get(indiceVerticeFinal);
    if(indiceVerticeInicial == indiceVerticeFinal) {
        matriz[indiceVerticeInicial][indiceVerticeInicial] = 1;
        verticeInicial.addGrau();
    } else {
        matriz[indiceVerticeInicial][indiceVerticeFinal] = 1;
        verticeInicial.addGrau();
```

```
            matriz[indiceVerticeFinal][indiceVerticeInicial] = 1;
            verticeFinal.addGrau();
        }
    }

    public List<Vertice> getAdjacencias(int indiceVertice) {
        int linha = indiceVertice;
        List<Vertice> adjacencias = new ArrayList<>();
        for(int j=0; j<vertices.size(); j++) {
            if(matriz[linha][j] == 1) {
                Vertice vertice = vertices.get(j);
                adjacencias.add(vertice);
            }
        }
        return adjacencias;
    }
```

Adicionar arestas no grafo é a operação mais básica que existe e é por onde devemos começar. Opera de forma posicional através de seus parâmetros para que possa criar a relação entre dois vértices na matriz de adjacência. Tanto esta classe quanto seus métodos trabalham de forma posicional se baseando em índices porque é assim que uma matriz funciona. Trabalhar com vértices em uma matriz de outra forma que não seja essa representa um erro no modelo de abstração da classe.

É necessário verificar se ambos os vértices são iguais (`indiceVerticeInicial = indiceVerticeFinal`), pois, como sabemos, caso essa assertiva seja verdadeira estamos lidando com uma situação de aresta recursiva e devemos agir de forma adequada. Caso os vértices não sejam iguais, estamos lidando com uma situação costumeira, onde necessitamos adicionar a aresta e aumentar em 1 o grau de um vértice de forma reflexiva.

Visto que já somos capazes de adicionar arestas entre dois vértices, podemos prover uma forma de recuperar essas conexões, em outras palavras, as adjacências. Para isto, o método `getAdjacencias(indiceVertice)` entra em ação. A partir de um índice, o qual representa a linha da matriz, todas as adjacências são

reunidas. Cada adjacência é representada por uma coluna que tenham o valor 1 em sua célula.

Por último, temos que tornar o grau de um vértice uma realidade em nosso projeto. Como um bom projeto de software, devemos saber separar bem as suas responsabilidades entre nossas classes. Então, métodos que operem o grau de um vértice devem ficar na já conhecida classe `Vértice` que criamos no capítulo 2, até porque isso é responsabilidade do vértice e não da matriz.

`src > main > java > grafo > core > Vertice.java`

```
void addGrau(){
    grau++;
}

public int getGrau() {
    return grau;
}
```

As finalidades de ambos os métodos `addGrau()` e `getGrau()` são bastante óbvias, mas é outra característica que deve chamar nossa atenção. O tipo de acesso a esses métodos difere, pois um opera em nível público (qualquer um pode acessá-lo) e outro opera em nível *default* (somente classes dentro do mesmo pacote que `Vertice` podem acessar este método). Esta escolha foi feita para fins de um bom design. Uma outra classe que não faça parte do pacote `core` da aplicação não pode acessar uma funcionalidade tão crítica como essa.

Precisamos agora de portas de entrada para tudo o que construímos, e não somente qualquer porta, mas uma que ofereça uma entrada limpa e concisa para as funcionalidades que serão expostas para o usuário final do nosso projeto. A única responsável por tal tarefa até o momento é a classe `Grafo` . Então é nela que devemos adicionar todos esses acessos.

`src > main > java > grafo > core > Grafo.java`

```
public Vertice getVertice(String rotulo) {
    this.existeVerticeOrThrow(rotulo);
    int indice = this.rotulosEmIndices.get(rotulo);
    return this.vertices.get(indice);
}

public void conectarVertices(String rotuloVerticeInicial, String
rotuloVerticeFinal) throws Exception{
    if(!this.existeVertice(rotuloVerticeInicial) ||
!this.existeVertice(rotuloVerticeFinal)) {
        throw new Exception("Para adicionar uma aresta ambos os vértices
devem existir.");
    }
    criarMatrizAdjacencia();
    int indiceVerticeFinal =
this.rotulosEmIndices.get(rotuloVerticeInicial);
    int indiceVerticeInicial =
this.rotulosEmIndices.get(rotuloVerticeFinal);
    this.matrizAdjacencia.adicionarAresta(indiceVerticeInicial,
indiceVerticeFinal);
}

public List<Vertice> getAdjacencias(String vertice) {
    this.existeVerticeOrThrow(vertice);
    int indiceVertice = this.rotulosEmIndices.get(vertice);
    return this.matrizAdjacencia.getAdjacencias(indiceVertice);
}

private boolean existeVerticeOrThrow(String vertice) {
    if(!existeVertice(vertice)) {
        throw new IllegalArgumentException("O vértice não existe.");
    }
    return true;
}

private boolean existeVertice(String rotuloVertice) {
    int indice = this.rotulosEmIndices.get(rotuloVertice);
    return this.vertices.get(indice) != null ? true : false;
}

private void criarMatrizAdjacencia() {
```

```
    if(this.matrizAdjacencia == null){
        this.matrizAdjacencia = new MatrizAdjacencia(new
ArrayList<Vertice>(this.vertices));
    }
}
```

Todos esses métodos satisfazem as condições mencionadas no parágrafo anterior, ou seja, são limpos e concisos. Isto acontece tanto através de nomes e parâmetros bem definidos que traduzem sua utilidade quanto a tratamentos de erro que deixam claro o que pode acontecer caso o uso não seja adequado. Mas somente isso não basta. Devemos nos preocupar com a qualidade da escrita empregada no código dentro de cada um deles. Para isso, analisemos cada um dos métodos.

O método `conectarVertices(rotuloVerticeInicial, rotuloVerticeFinal)` deixa bem claro qual é seu objetivo: conectar dois vértices por uma aresta, mas para isto precisa antes verificar se os vértices existem no grafo e criar a matriz de adjacência. Durante o processo de criação realizado pelo método `criarMatrizAdjacencia()`, é verificado se nenhuma matriz já foi criada por alguma solicitação de inclusão de aresta anterior pois, caso tenha sido, é esta matriz que deverá ser utilizada para futuras operações. Caso contrário, uma nova é criada e a operação de inclusão de aresta prossegue. Note o uso do atributo `rotulosEmIndices` para obter as posições dos vértices através de seus rótulos. Isso é necessário porque a matriz de adjacência funciona 100% posicional. Uma ressalva: não esqueça de criar o atributo `matrizAdjacencia`.

O método `getVertice(rotulo)` retorna um objeto do tipo `Vertice` dado um rótulo passado como parâmetro. Caso o vértice representado pelo rótulo não exista, uma exceção é lançada, informando que esta operação não é permitida.

O método `getAdjacencias(vertice)` funciona de forma similar ao `getVertice(rotulo)`. Retorna uma lista de vértices como adjacências de um determinado vértice e propaga uma exceção caso o vértice passado como parâmetro não exista.

E como usar tudo isso? Fácil. Crie um grafo, adicione alguns vértices, algumas arestas e pronto! Obtenha resultados.

```
Grafo grafo = new Grafo();

grafo.adicionarVertice("A");
grafo.adicionarVertice("B");
grafo.adicionarVertice("C");
grafo.adicionarVertice("D");

grafo.conectarVertices("A","B");
grafo.conectarVertices("A","C");
grafo.conectarVertices("A","D");

System.out.println("Grau do vértice A: " +
grafo.getVertice("A").getGrau());
System.out.println("Grau do vértice D: " +
grafo.getVertice("D").getGrau());
System.out.println("Grau do vértice C: " +
grafo.getVertice("C").getGrau());

System.out.println();
System.out.print("O vértice A possui as seguintes adjacências:");
List<Vertice> adjacentes = grafo.getAdjacencias("A");
for(Vertice vertice : adjacentes) {
    System.out.print(vertice.getRotulo() + " ");
}

System.out.println();
System.out.print("O vértice C possui as seguintes adjacências");
adjacentes = grafo.getAdjacencias("C");
for(Vertice vertice : adjacentes) {
    System.out.print(vertice.getRotulo() + " ");
}
```

Enfim, conseguimos uma representação computacional 100% fiel a todos os conceitos vistos e mais uma vez o nosso problema toma uma forma mais madura. Mas com esse amadurecimento mais questões aparecem: como realizar uma busca em um grafo às cegas, ou seja, sem conhecer o próximo vértice? A resposta desta

questão será de grande valia para a sua aplicação, visto que, à medida que sua empresa prospera e abre novas filiais, saber de cabeça quais delas estão conectadas, ou não, será uma tarefa árdua. Essa e outras questões estarão no próximo capítulo. Encontro você lá!

3.3 Conclusão

Entender conceitos como adjacência, incidência e grau de um vértice são fundamentais para o amadurecimento do conhecimento total sobre a Teoria dos Grafos. Eles formam o último passo básico, assim digamos, da nossa jornada. Então a partir daqui, veremos conceitos que utilizarão o que foi aprendido e implementado nos capítulos 2 e 3.

Entender os recursos e estruturas computacionais utilizados também é de grande importância, pois são eles que darão suporte às aplicações que um grafo pode ter. Diversificar o conhecimento através de vários conceitos, questionamentos e conclusões dá força à importância de todo o conteúdo visto neste capítulo, porque como já sabemos não existem soluções perfeitas. Então, quanto mais cartas postas na mesa forem conhecidas, melhor será.

Resumo

- A adjacência e a incidência de um vértice são conceitos diretamente ligados. A partir do momento em que um é determinado, o outro também é.
- Matrizes, no mundo da computação, são estruturas que as linguagens de programação oferecem para a manipulação de dados com duas dimensões.
- Para grafos não orientados, um vértice é adjacente a outro quando ambos são ligados entre si por uma aresta.

- Para grafos não orientados, é dita incidente toda aresta que vai em direção a um vértice.
- Em grafos não orientados, tanto as relações de adjacência quanto de incidência são mútuas, isto é, se o vértice A é adjacente a B, então B é adjacente a A. O mesmo vale para a incidência. Uma aresta *e* que conecte A e B é incidente tanto a A quando a B.
- Em dígrafos ou grafos orientados as relações de adjacência e incidência não são mútuas, elas dependem da direção da aresta.
- Para que uma matriz possa ser considerada uma matriz de adjacência, ela deve ser quadrada $n \times n$, onde n representa o número de vértices do grafo, e deve ser possível mapear para cada vértice todos os seus vértices adjacentes em suas correspondentes coordenadas na matriz.
- Matrizes de adjacência podem assumir os seguintes valores: 0, 1 e 2.
- Para cada 1 existente em uma matriz de adjacência, deve-se considerar que haja uma aresta que conecte dois vértices representados pela combinação linha-coluna. Para cada 0, deve-se considerar o oposto.
- O valor 2 somente deve ser considerado em caso de arestas paralelas que são adjacentes a um vértice.
- Para que uma matriz possa ser considerada uma matriz de incidência, deve ser $v \times e$, onde v representa os vértices, e e, as arestas do grafo. Deve ser capaz também de mapear para cada aresta todos os seus vértices incidentes em suas correspondentes coordenadas na matriz.
- Matrizes de incidência podem assumir os seguintes valores: 0, 1 e 2.

- Para cada 1 existente em uma matriz de incidência, deve-se considerar que haja um vértice com uma aresta incidente a ele, onde ambos são representados pela combinação linha-coluna. Para cada 0, deve-se considerar o oposto.
- O valor 2 somente deve ser considerado em caso de arestas recursivas.
- O grau de um vértice em um grafo consiste no número de arestas que são incidentes a ele.
- O somatório de graus dos vértices de qualquer grafo não orientado se dá pelo dobro do número de arestas.
- Para determinar o grau de um vértice em um grafo não orientado qualquer usando a matriz de adjacência é necessário fazer a contagem de arestas incidentes ao vértice, localizadas na coluna da matriz que representa o vértice.

Capítulo 4
Procurando uma agulha no palheiro

Alguma vez você já ouviu a expressão "procurar uma agulha no palheiro"? É muito comum ouvi-la quando estamos à frente de uma tarefa árdua e, no mundo dos grafos, isso não é diferente, até porque procurar um vértice em um grafo completamente desconhecido pode ser tão difícil quanto procurar uma agulha em um palheiro.

Mas por que tão difícil assim? Nos capítulos anteriores, já vimos formas de buscar vértices em um grafo, então a história deveria ser outra, certo? Errado. As buscas que utilizamos até agora não consideravam o fato de não se conhecerem as arestas que conectam os vértices.

Assim como Morpheus disse para Neo em Matrix, *free your mind*, libere sua mente de tudo o que está ao seu redor neste momento. Agora pense em um labirinto gigantesco daqueles que, só de olhar, já dão medo, onde é possível ficar perdido durante dias, e até talvez nunca encontrar uma saída. Desesperador, não? Eu, ao contrário, adoro labirintos. Acho que são estruturas formidáveis. Suas sinuosidades e meandros me fascinam. Mas, voltemos ao seu desespero.

Você deseja encontrar uma saída, obviamente, mas não faz a mínima ideia nem de como começar. Então você se lembra da estória de Teseu, herói grego mitológico, que obteve sucesso nesse mesmo problema enquanto estava atrás do terrível Minotauro. O conto nos diz que ele não conhecia o labirinto e, para não se perder, utilizou um novelo de linha de lã para lembrar do caminho que tinha percorrido. Essa solução deu certo porque ele logra matar o Minotauro e consegue sair de seu labirinto (Brandão, Junito de Souza; 1986).

Figura 4.1: Teseu luta contra o Minotauro em seu labirinto. Fonte: http://www.mitografias.com.br/2012/01/o-mito-de-teseu-parte-final-o-combate-contra-o-minotauro/

A chave para uma busca, às cegas, de um vértice em um grafo reside aí. Precisamos de uma forma de lembrar o caminho que fizemos, igual a Teseu com seu novelo de lã, para que não nos percamos. Que alívio que o Minotauro não está nos perseguindo.

Na Teoria dos Grafos existem duas técnicas amplamente conhecidas que contornam este problema: buscas em profundidade e buscas em largura. Ambas servem tanto para grafos não orientados quanto para orientados.

4.1 Buscas em profundidade

A técnica de busca baseada na profundidade (*Depth-First Search* em inglês) de um grafo é uma ideia que se baseia em percorrer um grafo totalmente a partir de um vértice inicial sem o conhecimento prévio de sua estrutura. Sua forma mais conhecida, citada em diversos livros e analisada nesta seção, é o *Algoritmo de Trémaux* (Even, Shimon; 1979).

Como vimos, Teseu não conhecia a estrutura do labirinto, ele não sabia para onde ir de cabeça. Logo, devemos analisar nossos grafos sob a mesma condição. Não conheceremos sua estrutura e consequentemente nenhum pré-planejamento é possível. As decisões de quais direções tomar devem ser feitas enquanto se percorre o grafo. É importante também que o algoritmo consiga realizar sua tarefa sem ficar vagueando inutilmente no grafo e perceba quando chegou ao fim (Even, Shimon; 1979).

Como estamos lidando com uma busca, esperamos encontrar algo, obviamente, e esse deve ser o ponto em que o algoritmo deve notar que chegou ao fim, ou seja, quando o vértice procurado for encontrado. Caso ele não seja, devemos prover alguma forma de sinalizar isso.

Devemos admitir que o uso do barbante por parte de Teseu foi uma ideia brilhante porque é ao mesmo tempo simples e genial. Como concluímos parágrafos antes, o ponto principal para uma busca eficaz é lembrar por onde passamos, então usaremos do mesmo artifício que Teseu. Marcadores nas passagens feitas nos ajudarão a reconhecer por onde devemos voltar e para onde ir (Even, Shimon; 1979). Entenda por passagem cada contato formado entre um vértice e uma aresta.

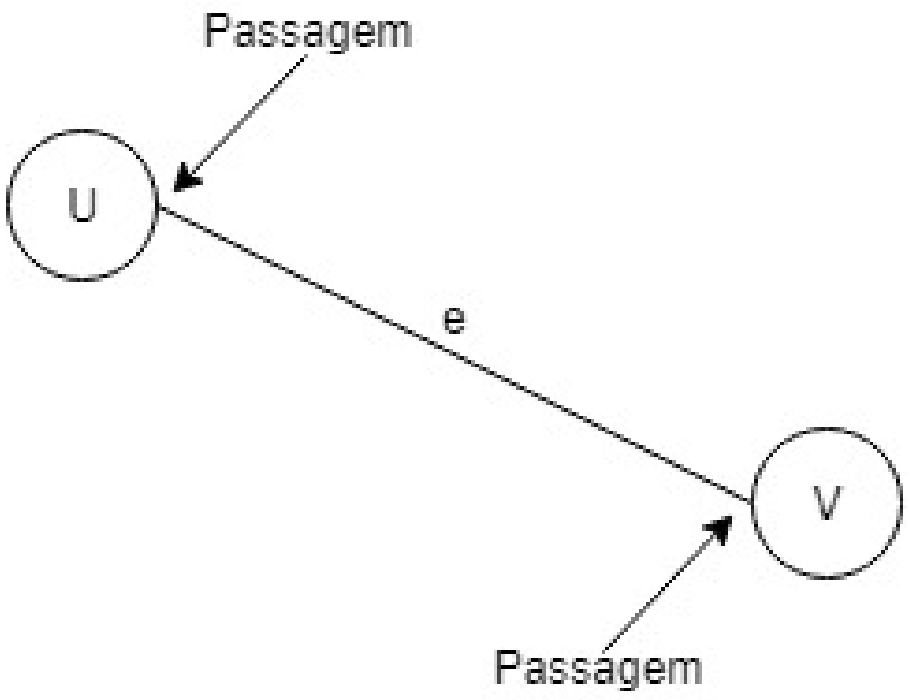


Figura 4.2: Dois vértices, uma aresta e suas duas passagens.

O grafo da imagem demonstra as passagens que dois vértices e uma aresta geram. Para arestas recursivas, a regra é a mesma, só que, neste caso, as duas passagens estarão no mesmo vértice.

Trabalharemos com dois tipos de marcadores, `P` e `S`, que representam, respectivamente, a *primeira* entrada em um vértice e a *saída* dele. No caso do vértice inicial, ele não terá passagens marcadas com `P` porque suas passagens serão usadas somente para saída. É comum encontrar em outras obras o uso da abordagem de aplicação de marcadores, exceto pelas letras que geralmente são conhecidas como `F` (de *first*) para a primeira entrada e `E` (de *exit*) para saída. Mas se você se sente mais confortável usando outras letras ou até símbolos, não se acanhe, o importante é usar o mecanismo de marcação que melhor lhe atende.

Independentemente da sua escolha, saiba que os marcadores não podem ser apagados nem movidos uma vez posicionados (Even,

Shimon; 1979). Com eles em mãos, devemos seguir as seguintes regras para realizar a nossa busca.

Regra 1

Escolha um vértice inicial.

Essa regra é a mais óbvia de todas, até porque todo labirinto tem uma entrada, logo um grafo também deve ter.

Regra 2

A partir do nó atual, pegue a próxima passagem não marcada, marque as duas passagens, e torne o nó em que você chegou o atual (Lafore, Robert; 2004).

Saber escolher a próxima passagem não marcada é muito importante, pois é aqui que aquelas decisões de direção que foram citadas no início do capítulo serão tomadas. Independente da direção seguida, sempre marque a passagem de saída tomada como `S`. Se o caminho feito o levar a um vértice novo (ainda não visitado), marque essa passagem de entrada como `P`. Mas se você acabar entrando em um vértice que já tem alguma passagem marcada como `P` (você já esteve ali em algum momento), marque essa passagem como `S`. O vértice inicial não possui passagens marcadas como `P` pois a partir dele você só vai sair.

Regra 3

Se você não pode mais executar a regra 2 é porque não existem mais passagens não marcadas, volte para onde estava e torne este nó o atual (Lafore, Robert; 2004).

Este é o momento em que você chegou a um beco sem saída. Então, o que fazer? Volte o caminho feito.

Regra 4

Se você não pode mais executar a regra 2 e a regra 3 é porque terminou (Lafore, Robert; 2004).

Esta regra representa o nosso ponto de parada, aquele momento em que a técnica usa de seus próprios mecanismos para notar que não deve mais percorrer o grafo. Os motivos pelos quais a parada é necessária se resumem a dois fatores: ou você encontrou o que precisava e não há mais a necessidade de continuar procurando, ou você já percorreu o grafo inteiro. Para o último fator, é conhecido que a busca sempre termina pelo vértice que a iniciou, isto é, você sairia do labirinto por onde entrou.

Apliquemos agora todas as regras aprendidas em um exemplo prático.

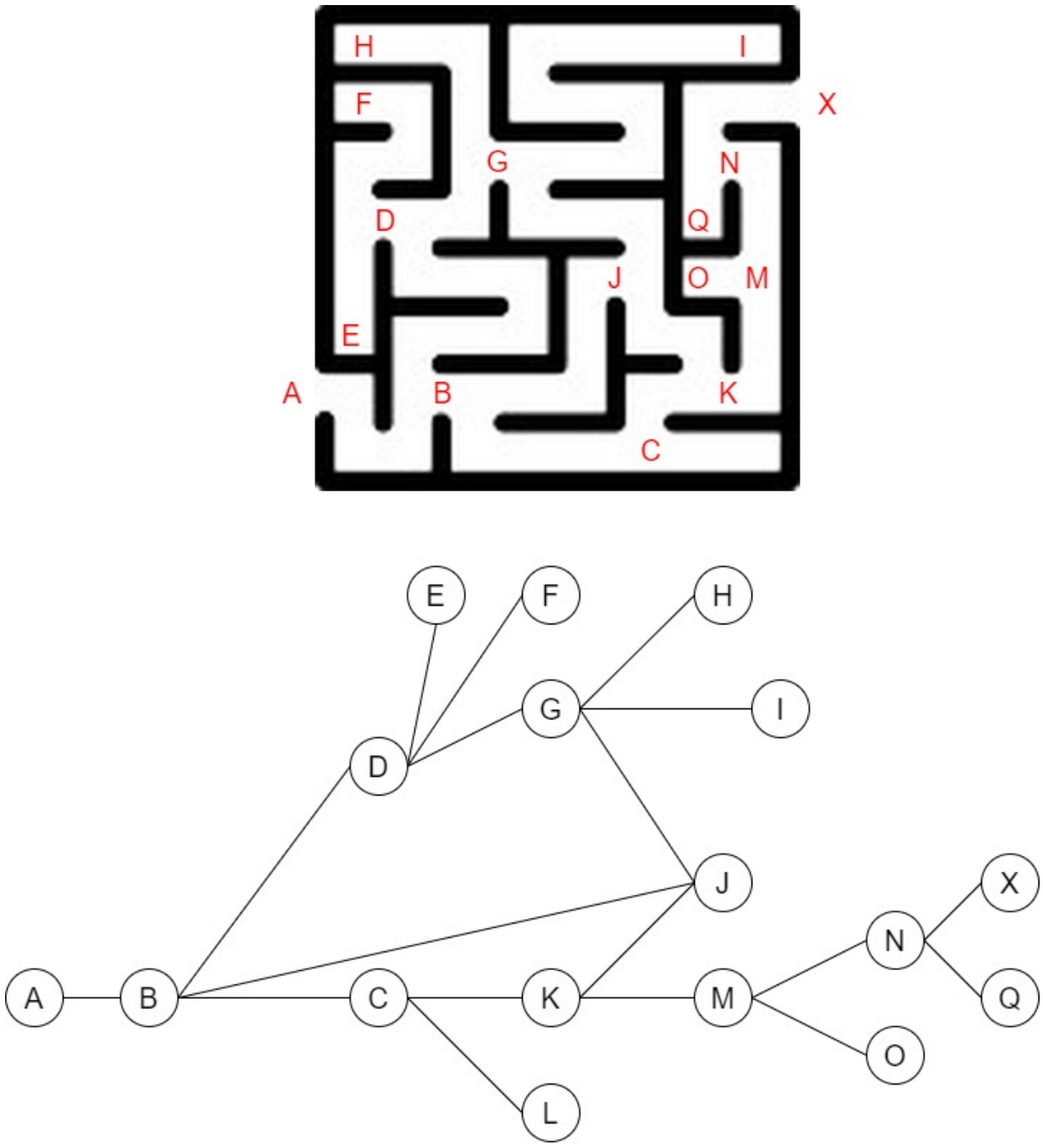


Figura 4.3: Labirinto e seu respectivo grafo.

CURIOSIDADE

Você sabia que é possível transformar um labirinto em um grafo? Pegue a entrada e a saída do labirinto e transforme os dois em vértices. A partir deles, encontre todos os pontos que formem ramificações no caminho e becos sem saídas. Cada um desses se transformará em um vértice. Agora, com todos os vértices mapeados, montar o grafo é um pulo. Ligue todos os vértices respeitando os caminhos do labirinto e, *voilà*, o seu grafo está pronto.

O grafo desta imagem é a representação do labirinto acima dele. O vértice A representa a entrada do labirinto e o vértice X representa a saída. O objetivo é a partir de A chegar a X, ou seja, você está procurando a saída do labirinto.

A essa altura, você já deve ter notado que a primeira regra já foi aplicada nesta situação, pois já foi determinado que o vértice A é o inicial. Então o que nos resta é seguir em frente e aplicar a regra 2, saindo de A em direção a B. No momento desta saída não podemos esquecer de realizar a marcação S na única passagem de saída de A, e aplicar P para B mas em sua passagem de entrada. Recomendo que você leia este e os parágrafos seguintes olhando a figura.

Aplicando a regra 2 novamente, vamos de B para D, marcando esta passagem de saída de B como S, e a de entrada de D como P. Em D continuamos aplicando a mesma regra e vamos para o vértice E, realizando o mesmo esquema de marcações: S para a passagem de saída de D, e P para a passagem de entrada de E. Mas em E não há mais para onde ir, chegamos a um caminho sem saída.

Para esses casos onde não é mais possível aplicar a regra 2, passamos para a seguinte e aplicamos a regra 3, e assim voltamos para o vértice D. Em D, mais uma vez constatamos que existem

outras passagens ainda não exploradas, portanto, regra 2, então vamos em direção ao vértice `F`, marcando esta passagem de saída de `D` como `S`, e a de entrada em `F` como `P`. Repetindo este processo até encontrar o vértice `X` teremos feito o caminho a seguir.

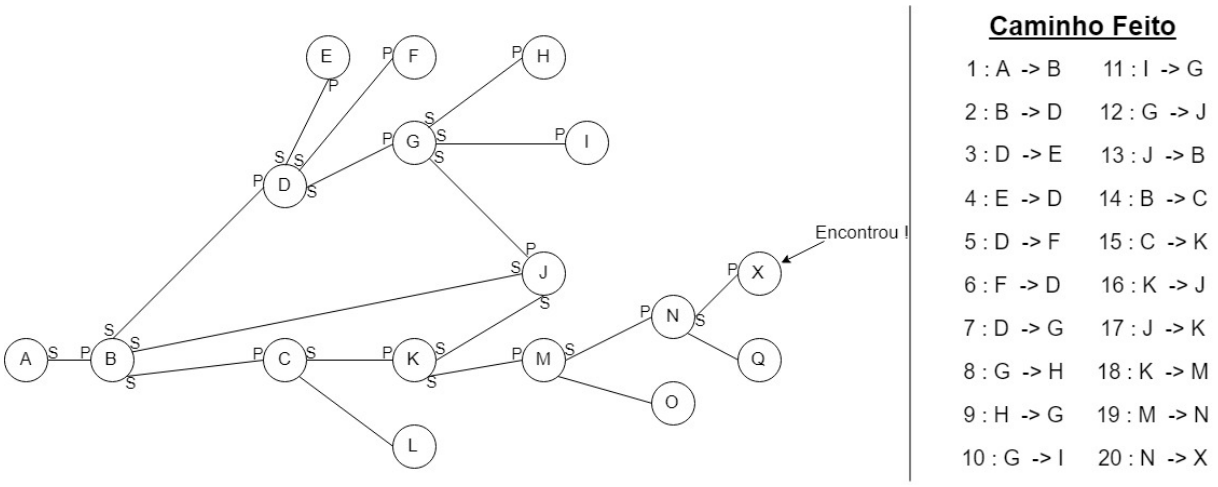


Figura 4.4: Aplicação do algoritmo de busca em profundidade no grafo do labirinto.

Um ponto interessante a ser percebido é o décimo terceiro passo da busca. Note que de `J` vamos para `B`, mas `B` já tinha sido visitado no passo 1. Então, qual marcação aplicar? Para sanar essa dúvida devemos nos lembrar dos conceitos básicos da técnica e da regra 2.

Marcadores não podem ser apagados ou movidos. Uma vez que entramos em um vértice pela primeira vez e marcamos essa passagem de entrada como `P`, outra não pode receber esse mesmo marcador.

Já a regra 2 nos diz que, para ir de um vértice a outro, devemos sempre pegar a próxima passagem não marcada do vértice em que estamos (o atual), e marcar as duas passagens dessa movimentação como `S`. Neste caso, elas são a de saída do nó atual, `J`, e a de entrada do vértice destino também, `B`, visto que já o visitamos no passado. Então por estes motivos o décimo terceiro passo da busca é justificado.

O passo a passo dos vértices percorridos para no vigésimo passo, de `N` para `X`, porque o vértice `X` é o que estamos procurando. Logo, não há necessidade de continuar.

Um trabalho escrito por brasileiros da Universidade Federal da Paraíba - UFPB mostra uma aplicação muito legal desta técnica. Através de algumas modificações, eles conseguiram desenvolver jogos de labirinto em um ambiente 3D, que tem seus próprios labirintos criados de forma aleatória, e com personagens que se locomovem por eles de forma autônoma. O trabalho é muito bom e o resultado final é muito interessante (Menezes Jr., Rômulo; Machado, Liliane; Medeiros, Álvaro; 2012). Para mais informações dê uma folheada até a seção de referências bibliográficas.

Eu, particularmente, tenho um passado com este algoritmo. Em setembro de 2018, estive em Campos do Jordão - São Paulo e fui ao parque Amantikir. Este parque é muito conhecido pelos seus diversos jardins e seus dois labirintos feitos de plantas, sendo um com paredes altas feito de alguma espécie de planta trepadeira e outro mais raso feito com alguma espécie de gramínea.

Eu e minha esposa fomos nos dois e obviamente nos perdemos no maior assim como outros turistas. Naquele momento, a primeira coisa em que eu pensei foi neste algoritmo e como podia aplicá-lo. Não tínhamos nenhuma linha ou barbante, então quando encontrávamos uma ramificação um ficava lá parado esperando e o outro percorria os vários caminhos até encontrar um novo caminho para seguir. No fim tudo saiu bem, continuamos o passeio e eu fiquei feliz por ter aplicado o algoritmo. A seguir, fotos dos labirintos.

Figura 4.5: O labirinto de paredes altas.

4.2 Buscas em largura

Inicio esta seção com uma pergunta. Você acreditaria se eu dissesse que, além de ser possível encontrar a saída de qualquer labirinto, é igualmente possível descobrir o menor caminho até lá? Provavelmente não, e não é necessário ser nenhum gênio para chegar a essa conclusão, até porque encontrar a saída de um labirinto às cegas já é algo, por si só, surreal. Mas acredite ou não, é possível, sim, determinar o menor caminho e é mais fácil do que você pensa.

Edward Moore, em seu estudo sobre caminhos em labirintos, publicou um trabalho em 1959 que propunha uma solução para o problema do menor caminho possível. Através de um simples algoritmo com alguns poucos passos, a obra de Moore ganhou um peso imenso, pois seu trabalho se tornou o que viria a ser a base para uma série de trabalhos futuros, proporcionando uma amplitude maior para a gama de aplicações que esse algoritmo pode ter (Even, Shimon; 1979).

Mas qual é a ideia que gira em torno do algoritmo de Moore que a torna tão simples e enxuta? Ao contrário de seu irmão, o algoritmo

de busca em profundidade, o algoritmo de busca em largura não explora um determinado caminho indo até o fim dele para, caso não ter encontrado uma saída, regredir e assim partir para o próximo caminho. Ele faz algo mais simples: trabalha com o conceito de ondas. Ondas? Mas o que seriam ondas em um grafo?

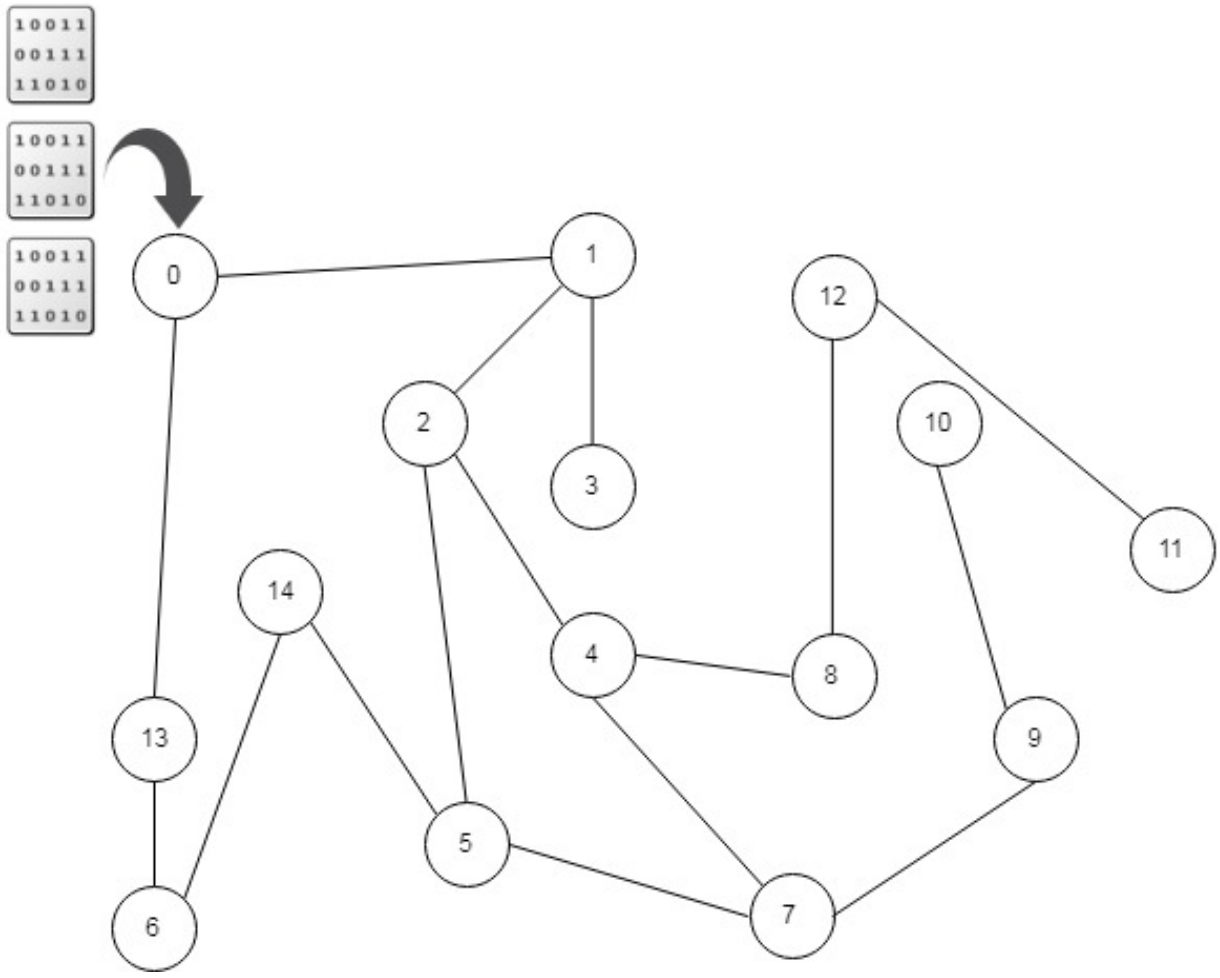


Figura 4.6: Grafo das conexões entre servidores de um jogo online

O relógio da cozinha marca exatamente duas da manhã e você está sentado em seu sofá sozinho, todos estão dormindo. É madrugada e não há ninguém acordado para o atrapalhar agora. Só existem na sala de estar os seguintes elementos: você, o videogame e a televisão. É o momento perfeito para aquela jogadinha noturna. Sem mãe, pai, esposa, marido, filho, filha, cachorro ou periquito para dizer que você está perdendo o seu tempo.

Durante a partida, lá para as 04:30 AM, você se dá conta de que as informações que são geradas enquanto você joga devem ser transmitidas para todos os outros participantes em tempo real, já que é *online*. Então começa a se perguntar. Como a informação consegue ir tão rápida de um ponto a outro sendo que existem vários servidores do seu jogo espalhados pelo mundo inteiro? Como os dados gerados sabem o menor caminho que devem percorrer até o servidor correto? Neste cenário passar pela menor quantidade de servidores possíveis é o menor caminho.

Como um bom estudante da teoria dos grafos, você já estava desconfiando de que, para esta situação, mapear os servidores e as conexões entre eles é a chave para formar, respectivamente, os vértices e as arestas de um grafo. Então, sem perder tempo, faz uma pesquisa na internet e consegue encontrar as respostas necessárias. O resultado final é o grafo da imagem anterior.

Agora, em relação ao problema do menor caminho, você também sabe que o algoritmo da busca em largura é a resposta. Então, engajado em resolver o enigma, determina o ponto de onde esses dados são transmitidos (ponto de entrada) e para qual ponto eles devem chegar (ponto de saída). Isso, no fim das contas, se traduz em informações sobre a partida em curso entrando pelo vértice `0` e tentando encontrar seu caminho até o vértice `3`.

Contudo, note que existem vários caminhos possíveis de `0` até `3` mas dois em especial chamam a sua atenção, são eles: `0 -> 1 -> 3` e `0 -> 13 -> 6 -> 14 -> 5 -> 2 -> 1 -> 3`. Mais uma vez, não é necessário ser um gênio para saber que o primeiro caminho é a solução ótima para o problema. Mas como o algoritmo chega a tal conclusão? Através de um conjunto de fatores. Um deles já conhecemos, são as ondas, outro, também conhecido, a ideia de pôr marcadores em vértices já visitados e, por último, o uso de uma função que realiza o cálculo da distância (Even, Shimon; 1979).

A essa altura da leitura você já deve estar irritado porque ainda não sabe o que são as tais misteriosas ondas das quais tanto falo.

Então, sem mais delongas, observe atentamente mais uma vez o grafo da imagem anterior. Ondas são nada além de uma outra forma de se estabelecer uma ordem de visitar vértices. Qual é a ordem usada para visitar vértices em uma busca em profundidade? Visite o **próximo** vértice adjacente ainda não visitado, resumindo de uma forma geral, e a partir deste repita o mesmo processo, obviamente com ressalvas. Agora, a ordem não é mais essa, e sim: dado um determinado vértice, visite **todos** os seus vértices adjacentes ainda não visitados, e partir deles repita o processo respeitando as ressalvas existentes.

Essas são as ondas. A imagem a seguir torna tudo mais claro.

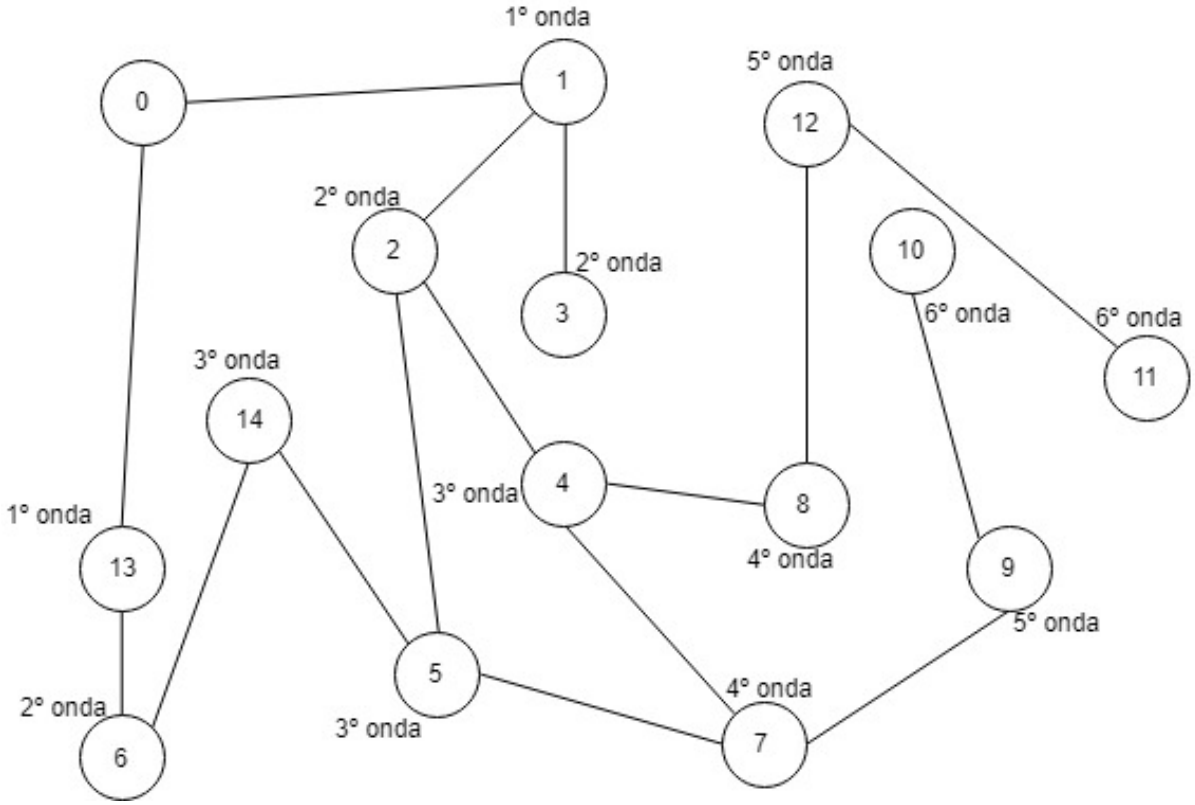


Figura 4.7: As ondas de um grafo

Cada descrição colada nos vértices representa a ordinalidade da onda e nos mostra os vértices que devem ser analisados. Os vértices `1` e `13` formam a primeira onda porque são adjacentes ao vértice inicial, `0`. Já `6`, `2` e `3` formam o conjunto da segunda onda

pelo mesmo motivo, são adjacentes aos vértices da primeira onda, e assim por diante pela mesma regra até não sobrarem mais vértices.

Esse comportamento pode ser sintetizado em uma função `w(i)`, onde `i` assume um valor maior ou igual a zero e representa uma determinada onda. O resultado final gerado por esta função é o conjunto de vértices pertencentes àquela onda (Even, Shimon; 1979). Então, `w(1)` gera os vértices da primeira onda, `w(2)` os da segunda onda, e assim sucessivamente.

Então, para a última imagem, quais são os resultados da aplicação da função `w(i)` ?

- w(0) = {0}
- w(1) = {1, 13}
- w(2) = {2, 3, 6}
- w(3) = {4, 5, 14}
- w(4) = {7, 8}
- w(5) = {9, 12}
- w(6) = {10, 11}

Conforme veremos mais à frente, o entendimento dessa função será de suma importância para o entendimento do algoritmo proposto por Moore. Contudo, ainda existem algumas perguntas não respondidas. Por que `i=0` não representa a primeira onda, e sim, `i=1` ? E, por que o vértice `14` não faz parte da quinta onda (`i=4`) já que o vértice `5` faz parte da quarta (`i=3`) e eles são adjacentes entre si?

Se formos seguir a regra à risca, de fato `i=0` deveria representar a primeira onda, em vez de `i=1` . Mas isto não acontece dessa forma porque calcular `w(i)` quando `i=0` não é de valia alguma para o problema que precisa ser resolvido, visto que o resultado da função para este valor de `i` sempre resulta no vértice inicial, nosso ponto de partida.

Para a segunda pergunta, a resposta reside em um detalhe já citado. O esquema de varredura usado sempre procura por vértices

adjacentes ainda não visitados, e neste caso específico o vértice 14 já foi visitado pois é adjacente ao 6. Logo, quando o vértice 5 consulta seus adjacentes e nota que 14 é um deles, verifica em seguida se ele já foi visitado, porque, se ele já foi, estará marcado e inserido em algum conjunto proveniente de uma onda anterior.

Note que eu falei sobre marcação de vértices. Este tema não é novo, já passamos por ele na seção anterior. Aqui ele também é utilizado, de uma forma diferente, mas sob a mesma circunstância, isto é, a ausência de pré-planejamento sobre o trajeto a ser feito e de conhecimento sobre a estrutura do grafo.

Até este exato ponto já cobrimos vários assuntos: o esquema de exploração usado (as ondas); a função que resume este comportamento; o uso de marcadores e curiosidades que surgiram. Mas, o ponto principal de todo o nosso questionamento ainda não foi solucionado. Como o algoritmo proposto por Moore consegue definir o menor caminho de um ponto a outro?

Falar de menor caminho sem falar distância é algo inviável. Então é esse o grande X da questão. Como construir uma função, a qual denota um comportamento, que consegue calcular a distância de um ponto a outro? E não somente de um ponto a outro, eis um fato novo aqui, o algoritmo de Moore consegue determinar a menor distância do vértice inicial para **todos** os outros vértices do grafo desde que estejam conectados.

Voltando à pergunta do parágrafo acima, é simples! Se eu sei que de um vértice A para um vértice B, a distância é de uma aresta, então eu já calculei o menor caminho possível entre esses dois vértices. Contudo, esse valor não deve ser desprezado, e sim, deve ser utilizado para explorar os vértices adjacentes a B ignorando A.

Analisando B, chego à conclusão de que a distância também é de uma aresta, mas a partir dele até o vértice C. Logo, posso afirmar que a distância de A para C são de duas arestas, porque de A para B é 1, de B para C também é 1 e $1 + 1 = 2$. É assim que a

função **d** funciona. Ela, de uma forma geral, representa o menor caminho possível entre dois vértices, ou seja, `d(v) = d(s,v)` (Even, Shimon; 1979).

Complicou tudo agora, não é? Imagino um nó sendo dado na sua cabeça. Calma, vamos descomplicar.

Quando digo `d(v) = d(s,v)`, quero dizer que o menor caminho possível do vértice inicial do grafo, `s`, até um vértice `v` qualquer é a distância, representada pela função `d`, entre os vértices `s` e `v`. Viu?! Não é tão difícil assim.

Entender o que a função `d` representa é entender a união de todos os fatores apresentados até agora. Inicialmente precisávamos de um vértice inicial, o qual seria o nosso ponto de partida. Em seguida, um esquema de varredura eficiente, sendo sucedido por uma função que consegue mapear como essa varredura acontece e um esquema de marcações que garante que não checamos o mesmo vértice mais uma vez. Por último, precisávamos de um marco final que resolveria completamente o problema proposto, eis aí a função `d`. Agora, nossa mão está repleta de ases e estamos prontos para bater a mesa. O que falta é entender o algoritmo.

O algoritmo parte dos seguintes princípios:

- Inicialmente todas as distâncias a partir do vértice inicial são infinitas.
- A distância do vértice inicial é sempre 0.

Ambas as afirmações fazem total sentido. Para a primeira, já sabemos que não há conhecimento prévio da estrutura do grafo, logo não há como deduzir um valor concreto. E, para a segunda, a distância de um ponto para ele mesmo é 0 porque não há distância (Even, Shimon; 1979).

Para dar apoio ao esquema de varredura em ondas e de verificação de vértices marcados, o algoritmo usa uma fila e um *array*, respectivamente (Even, Shimon; 1979). Filas representam um

conceito diferenciado sobre a manipulação de elementos em um conjunto. Por sua vez, um conjunto representa um aglomerado de dados, como os *arrays*. Então uma fila é um conceito que pode ser implementado em um *array*, o qual é, de fato, uma estrutura de dados. Quando me refiro à manipulação, quero dizer a ordem como elementos são adicionados, removidos e deslocados após cada operação. A que usaremos possui o esquema FIFO (*first-in-first-out*), isto é, os primeiros elementos a serem adicionados serão os primeiros a ser removidos, assim como uma fila de banco (Lafore, Robert; 2004).

O algoritmo adiciona o vértice inicial à fila pois, como sabemos, `w(0)` sempre resulta no vértice inicial, e o adiciona também ao *array* de vértices marcados para que não o visitemos novamente. Feito isso, ele retira o primeiro vértice da fila, neste momento o vértice inicial, e analisa suas adjacências. Para cada adjacência não marcada, ou seja, um vértice novo, ele o adiciona no *array* de vértices marcados, na fila de vértices pertencentes à onda (neste momento `w(1)`), e computa sua distância a partir do vértice inicial.

Este processo se repete até que a fila esteja completamente vazia, após todos os vértices terem sido analisados. Após isso, o algoritmo chega ao fim. Caso um vértice adjacente seja analisado e já estiver marcado, o algoritmo o ignora e continua procurando por adjacências não marcadas ainda. E é assim que o algoritmo inicialmente proposto por Moore funciona.

Eu disse que era simples, não disse?

Além do exemplo da seção anterior, do trabalho desenvolvido por brasileiros que utiliza ambas técnicas de busca apresentadas, aqui também temos como amostra o trabalho desenvolvido na universidade do estado de Santa Catarina, envolvendo o jogo "resta um". Consiste em um estudo comparativo onde são analisadas diversas técnicas de busca, inclusive as apresentadas aqui, para concluir, ao final, qual delas traz a maior quantidade de resultados

para o jogo no menor tempo possível (Ochner, Anderson; Pezzini, Anderson; 2015).

É bem interessante perceber como a gama de aplicações possíveis é rica porque tal fato une dois pontos importantes, a teoria à prática.

4.3 Dando vida às buscas

Busca em profundidade

No capítulo anterior, construímos o que entendemos como a estrutura que nos daria o devido suporte à montagem das conexões de um grafo e assim nasceu a matriz de adjacência. Nesta seção utilizaremos tudo que há para dar impulso à implementação das buscas.

Para fins de organização do código-fonte do nosso projeto criaremos estruturas apartadas das convencionais, isto é, novas classes. Então, no pacote `search`, crie a classe `Caminho`.

```
src > main > java > grafo > search > Caminho.java
```

```
package main.java.grafo.search;

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

class Caminho {
    private Map<String, String> caminho;

    public Caminho() {
        this.caminho = new HashMap<>();
    }
```

```
    void ligar(String anterior, String proximo){
        this.caminho.put(anterior, proximo);
    }

    public List<String> gerar(String origem, String destino){
        List<String> resultado = new ArrayList<>();
        String no = destino;
        while(no != origem && this.caminho.containsKey(no)) {
            resultado.add(no);
            no = this.caminho.get(no);
        }
        resultado.add(no);
        Collections.reverse(resultado);
        return resultado;
    }
}
```

Não se preocupe agora com esta classe, tudo fará sentido mais à frente. Somente se lembre de que ela será tanto **importante** para a busca em profundidade quanto para a busca em largura.

Crie, no mesmo pacote, a classe `BuscaEmProfundidade`.

```
src > main > java > grafo > search > BuscaEmProfundidade.java

package main.java.grafo.search;

public class BuscaEmProfundidade { }
```

Para que possa ser instanciada utilizaremos de um mecanismo *singleton* muito conhecido porque nos trará muita agilidade no uso. Adicione o código a seguir junto e realize todos os *imports* necessários para que possamos partir para o que será o cerne de todo o desenvolvimento.

```
src > main > java > grafo > search > BuscaEmProfundidade.java

package main.java.grafo.search;

public class BuscaEmProfundidade {
```

```
    private static BuscaEmProfundidade instance;

    private BuscaEmProfundidade(){ }

    public static BuscaEmProfundidade getInstance() {
        if(instance == null) {
            return new BuscaEmProfundidade();
        }
        return instance;
    }
}
```

Aqui está o tesouro pelo qual estamos procurando há tanto tempo. Este é o método que representa e executa a busca em profundidade. Tudo em sua volta serve para dar suporte a ele.

src > main > java > grafo > search > BuscaEmProfundidade.java

```
package main.java.grafo.search;

import java.util.LinkedHashSet;
import java.util.List;
import java.util.Stack;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Vertice;

public class BuscaEmProfundidade {

    /*instance*/
    /*construtor privado*/
    /*método getInstance()*/

    public List<String> buscar(Grafo grafo, String origem, String destino)
{
        Stack<String> roloDeBarbante = new Stack<String>();
        LinkedHashSet<String> verticesVisitados = new
LinkedHashSet<String>();
        Caminho caminho = new Caminho();

        roloDeBarbante.push(origem);
```

```
        while(!roloDeBarbante.empty()){
            String v = roloDeBarbante.pop();
            if(v.equals(destino)) {
                break;
            }
            for(Vertice u : grafo.getAdjacencias(v)) {
                String rotulo = u.getRotulo();
                if(!verticesVisitados.contains(rotulo)) {
                    verticesVisitados.add(rotulo);
                    caminho.ligar(rotulo, v);
                    roloDeBarbante.push(rotulo);
                }
            }
        }

        return caminho.gerar(origem, destino);
    }
}
```

O método `buscar` recebe o grafo no qual vai realizar a busca e os vértices de origem e destino como parâmetros, e devolve um caminho de vértices visitados em forma de lista, onde este caminho é gerado pela classe `Caminho`.

Lembra do novelo de lã usado por Teseu em sua empreitada contra o Minotauro? Aqui ele toma forma de uma pilha chamada de `roloDeBarbante`. Uma pilha é, assim como uma fila, uma ideia de como organizar itens dentro de um *array*. A pilha preza que o primeiro item a entrar será o último a sair, de forma semelhante a uma pilha de pratos. Não podemos tirar o prato que está no fundo da pilha, somente o que está em seu topo. A estrutura de dados funciona exatamente da mesma forma (Lafore, Robert; 2004).

O nome `roloDeBarbante` pode não ser um dos mais intuitivos do mundo para dizer que esta variável é uma pilha, mas achei interessante deixar esse nome assim para que você possa se lembrar da história mitológica e relacionar o seu uso.

Então o processo de busca começa, adicionamos o vértice de origem a pilha `roloDeBarbante`. Mantendo a analogia com Teseu e o labirinto do Minotauro, este momento seria visualizado como ele entrando no labirinto e largando ali uma ponta do rolo do barbante para que não se perca.

Após, de forma repetitiva, verificamos se a pilha não está vazia para cada vez obter o próximo vértice que deve ser analisado (neste momento Teseu anda pelo labirinto). Nesse exato momento obtemos o vértice de origem que acabamos de adicionar à pilha.

Verificamos se este vértice analisado é igual ao vértice de destino porque caso seja chegamos ao fim e o algoritmo pode ser interrompido. Caso não seja continuamos. Para este vértice analisado obtemos seus vértices adjacentes e para cada um verificamos se ele já foi visitado anteriormente. Se não foi visitado ainda o marcamos como visitado, registramos esse caminho e adicionamos esse vértice à pilha. O registro desse caminho é feito pelo método `ligar(anterior,proximo)` da classe `Caminho`.

Todo o procedimento citado nos dois parágrafos acima se repete até que o vértice informado como destino tenha sido encontrado. E, é nesse exato momento que a classe `Caminho` ganha uma grande importância, pois quando o método `ligar` é chamado ele mantém uma trilha inversa usando o vértice adjacente como anterior ao vértice removido/recuperado da pilha. Parece confuso, não!?

Em outras palavras, se `caminho` devesse manter a trilha `A -> B -> C` na verdade ele manterá `C -> B -> A` e aí você se pergunta: por que complicar o que já está difícil? Fique calmo, entenderemos mais à frente.

Quando todo esse processo repetitivo acaba e chega a hora de retornar o caminho feito, o método `gerar(origem, destino)` é chamado. Este método cria uma lista a partir da trilha inversa computada. A lista representa o caminho da forma que esperávamos que fosse. Para construir esse caminho o método

`gerar` vai andando para atrás na trilha inversa a partir do vértice de destino até chegar ao vértice de origem. Uma vez que chega à origem, ele reverte o caminho feito e aí obtemos o resultado que esperamos.

Busca em largura

Muito do que entendemos e criamos na subseção anterior será ou utilizado aqui, ou copiado, então mais da metade do caminho já está andado. O que nos resta então é criar o que virá a ser o cérebro por trás de toda esta operação. Este será o nosso foco.

Crie a classe `BuscaEmLargura` e dentro dela adicione todo aquele mecanismo *singleton*. Sua classe deve se parecer assim.

```
src > main > java > grafo > search > BuscaEmLargura.java
```

```
package main.java.grafo.search;

public class BuscaEmLargura {

    private static BuscaEmLargura instance;

    public static BuscaEmLargura getInstance() {
        if(instance == null) {
            return new BuscaEmLargura();
        }
        return instance;
    }
}
```

Feito isso, podemos iniciar o entendimento do método `buscar`.

Diferentemente da busca em profundidade, esta usa uma fila FIFO, já explicada, representada pela variável `roloDeBarbante`. Este é um detalhe **MEGA** importante. Note que ambas as buscas possuem essa variável com o mesmo nome, e não é para menos pois, mesmo uma sendo uma pilha e outra, uma fila, nos dois casos

representam a mesma coisa: uma forma de **registrar o caminho percorrido**.

O código da busca em largura é praticamente igual ao de profundidade, exceto pelo uso da fila e seus métodos. A fila é o diferencial desse método de busca porque é ela que proporciona o efeito das ondas já estudado. Na técnica de profundidade, a pilha possui a mesma responsabilidade, é ela que permite que a busca aconteça de forma profunda no grafo e não larga.

Devido a tal fato, explicações rebuscadas não são necessárias. Então, que cessem os rodeio, adicione o método `buscar(grafo, origem, destino)` abaixo na classe `BuscaEmLargura` .

`src > main > java > grafo > search > BuscaEmLargura.java`

```
package main.java.grafo.search;

import java.util.LinkedHashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Vertice;

public class BuscaEmLargura {

    /*instance*/
    /*construtor privado*/
    /*método getInstance()*/

    public List<String> buscar(Grafo grafo, String origem, String destino)
{
        Queue<String> roloDeBarbante = new LinkedList<String>();
        LinkedHashSet<String> verticesVisitados = new
LinkedHashSet<String>();
        Caminho caminho = new Caminho();

        roloDeBarbante.add(origem);
```

```
        while(!roloDeBarbante.isEmpty()){
            String v = roloDeBarbante.poll();
            if(v.equals(destino)) {
                break;
            }
            for(Vertice u : grafo.getAdjacencias(v)) {
                String rotulo = u.getRotulo();
                if(!verticesVisitados.contains(rotulo)) {
                    verticesVisitados.add(rotulo);
                    caminho.ligar(rotulo, v);
                    roloDeBarbante.add(rotulo);
                }
            }
        }

        return caminho.gerar(origem, destino);
    }
}
```

Pronto! A classe `BuscaEmLargura` está terminada.

Pondo para funcionar e colhendo resultados

Altere o trecho do capítulo 3 onde foi mostrado como usar o que foi desenvolvido para se parecer com a amostra de código a seguir.

```
Grafo grafo = new Grafo();

grafo.adicionarVertice("A");
grafo.adicionarVertice("B");
grafo.adicionarVertice("C");
grafo.adicionarVertice("D");
grafo.adicionarVertice("E");
grafo.adicionarVertice("F");
grafo.adicionarVertice("G");
grafo.adicionarVertice("H");
grafo.adicionarVertice("I");

grafo.conectarVertices("A","B");
grafo.conectarVertices("A","C");
```

```
grafo.conectarVertices("A","D");
grafo.conectarVertices("B", "F");
grafo.conectarVertices("B", "I");
grafo.conectarVertices("D", "E");
grafo.conectarVertices("D", "I");
grafo.conectarVertices("D", "G");
grafo.conectarVertices("I", "A");
grafo.conectarVertices("I", "D");
grafo.conectarVertices("I", "C");
grafo.conectarVertices("I", "H");
grafo.conectarVertices("E", "A");

List<String> caminho = BuscaEmProfundidade.getInstance().buscar(grafo,
"D", "H");
System.out.print("Caminho feito por uma busca em profundidade: ");
for(String passo : caminho) {
    System.out.print(passo + " ");
}

caminho = BuscaEmLargura.getInstance().buscar(grafo, "B", "G");
System.out.println();
System.out.print("Caminho feito por uma busca em largura: ");
for(String passo : caminho) {
    System.out.print(passo + " ");
}
```

Espere o seguinte resultado:

```
Caminho feito por uma busca em profundidade: D I H
Caminho feito por uma busca em largura: B A D G
```

Modifique os parâmetros passados para os métodos de busca por outros valores, utilize vértices que não existem ou defina o ponto de chegada como o mesmo de partida e veja como o programa se comporta.

Feito tudo isso, você já pode pensar em um projeto mais completo para o seu chefe, inclusive com mais de uma forma de explorar as conexões entre as filiais da empresa. Mas e se ele solicitasse uma solução para redução de custos? Algo do tipo: "Acabei de receber

uma ligação importante da diretoria e precisamos reduzir o número de conexões entre as filiais porque o custo de envio de documentos está muito alto. Precisamos de uma forma de conectar todas, mas gastando o menos possível, mesmo que isso resulte em uma demora na chegada de documentos em uma filial.". Como você sairia dessa? Para sua sorte existem árvores que podem nos ajudar.

Sim, você não leu errado, está escrito "árvores" no parágrafo anterior. Será que vamos começar a falar de biologia e deixar os grafos de lado? As respostas para todas estas dúvidas, inclusive a solução para o seu chefe, se encontram no próximo capítulo. Já estou lá esperando você.

4.4 Conclusão

Buscas são de fato ferramentas importantes. Sem elas, estaríamos perdidos pois, dependendo do problema abordado, o grafo resultante de uma análise pode tomar medidas astronômicas. Utilizando um simples conjunto de regras estabelecido através um ponto de vista (profundidade ou largura) de como se explorar um grafo, é possível obter informações que antes eram desconhecidas.

Um ponto em comum entre as duas perspectivas de busca é a marcação do caminho feito. Esta técnica, usada desde que o mundo é mundo para não se perder em caminhos, nos facilita em seguir adiante ou voltar em diversos momentos. Mas somente isso não adianta se não possuirmos estruturas de dados que forneçam este suporte às nossas implementações. Obviamente, existem perguntas ainda sem respostas e para elas adquirimos novas ferramentas, mas este capítulo preenche nosso canivete suíço com peças importantes nesta jornada.

Resumo

- Até o capítulo anterior, tínhamos somente a noção de quais vértices compunham nosso grafo e para cada um deles qual era o seu grau e seus vértices vizinhos. Mas isso não era o bastante para que pudéssemos caminhar livremente de um vértice a outro.
- Buscas nos fornecem passos estruturados e bem definidos de como chegar de um vértice a outro sem conhecer o que há no meio entre eles dois.
- Buscas utilizam marcadores para que possam se lembrar quais vértices já visitaram.
- Existem diversos tipos de buscas, entre elas as em profundidade e em largura.
- Buscas em profundidade possuem uma abordagem de exploração mais intrusiva na estrutura do grafo. A partir de um vértice inicial, tenta-se sempre descobrir um vértice adjacente e não visitado ainda e a partir dele repetir o mesmo processo e assim sucessivamente. Caso não seja possível, se volta um vértice e se tenta executar o mesmo processo em uma outra ramificação. No final, a busca vai parar no vértice que a iniciou.
- Buscas em largura possuem uma abordagem de exploração mais abrangente. A partir de um vértice inicial, são visitados todos os vértices adjacentes e não visitados ainda. Após isso, para cada vértice adjacente é repetido o mesmo processo e assim consecutivamente, até não haver mais vértices adjacentes e não visitados. Esta busca fornece uma resposta para o problema do menor caminho entre vértices.
- Buscas em profundidade utilizam uma pilha para seu esquema de marcações, e buscas em largura uma fila FIFO.
- Tanto pilhas como filas são esquemas de organização de dados, onde ambas utilizam um *array* como estrutura

subjacente. Elas determinam como elementos serão adicionados ou removidos de um *array*.

Capítulo 5
Uma floresta à frente

No capítulo anterior, vimos que você tem um desafio à frente. Como minimizar todas as conexões entre filiais de forma que a empresa gaste o menos possível? A princípio, parece impossível desenvolver um mecanismo que consiga resolver este problema de forma automática, como uma ferramenta. Digo "automática" porque este trabalho é normalmente muito cansativo para ser realizado de forma manual, ele é complexo. Em outras palavras, não podemos eliminar as arestas do grafo uma a uma. Precisamos construir uma ferramenta que "leia" o nosso grafo e proponha uma solução para ele.

Esta necessidade é justificável porque atualmente sua empresa possui 8 filiais em todo o país e já sabemos que para esta configuração há uma certa complexidade. Imagina quando ela crescer para 10, 15 ou até 80 filiais no país inteiro? Quando isto acontecer, o que era difícil vai se tornar, de fato, impossível. Encontrar a solução agora é um fator decisivo.

Vimos também que parte da solução seria o uso de árvores, mas o que são e onde vivem? O termo "árvore" deve, de fato, causar alguma estranheza visto que a imagem que nos vem à mente é a de uma planta. Entretanto, não estamos nos referindo a esse tipo de árvore, e sim a uma estrutura que nos oferecerá uma nova perspectiva sobre um grafo. É essa perspectiva que será explorada nas seções adiante.

5.1 Árvores

A esta altura já vimos que grafos podem adotar infinitas formas dependendo do contexto em que estão. Alguns com poucos vértices e arestas, outros com muitos. Mas, independentemente do tamanho ou do contexto, pode um grafo ser uma árvore? Existe algum cabimento para esta afirmação?

Esqueçamos por um momento as árvores, precisamos antes entender um conceito ainda mais primordial. O que vem à sua cabeça quando ouve a palavra "ciclo"? O dicionário define ciclo como algo que parte de um ponto inicial e termina com a recorrência deste, em outras palavras, algo que termina onde iniciou. Este é um conceito mais abrangente, mas investigando sua aplicação em certos segmentos do conhecimento humano, tais como astronomia, física, matemática, podemos notar que ele acaba também dando uma ideia de repetição.

Nitidamente, podemos concluir que ciclos fazem parte do nosso cotidiano e para a teoria dos grafos isso não seria diferente. Quando dizemos que um grafo tem um ciclo, queremos dizer, na prática, que de um vértice a outro existe mais de 1 caminho (Harary, Frank; 1970). Você se lembra do grafo a seguir?

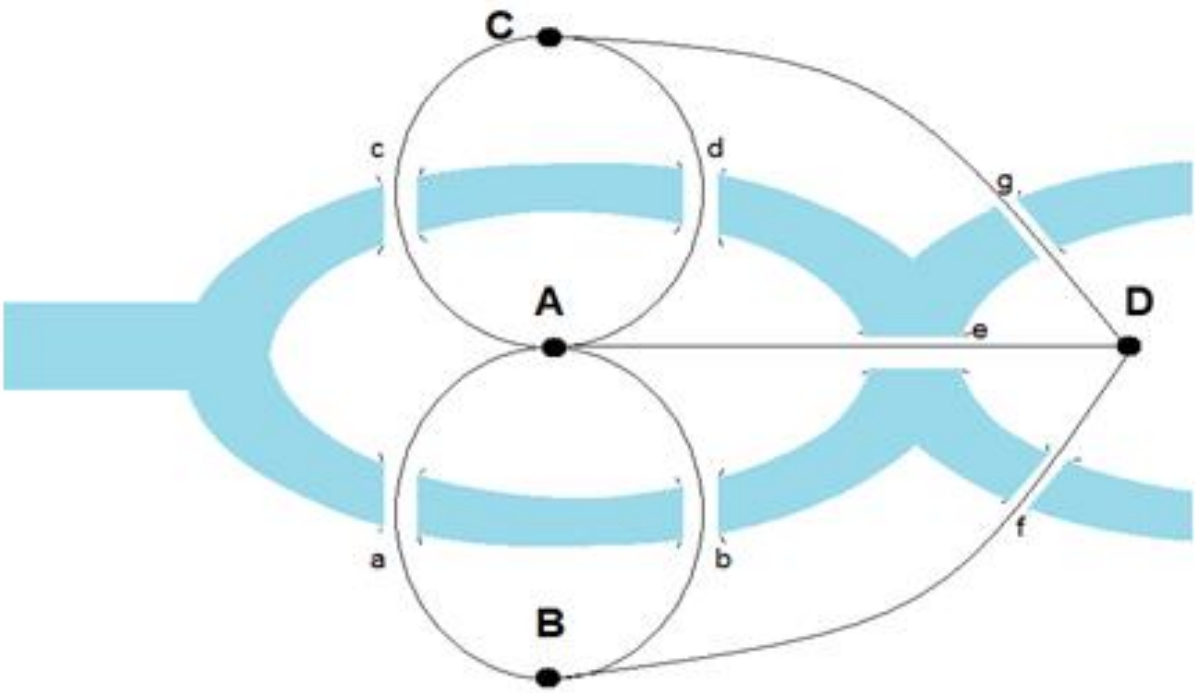


Figura 5.1: Grafo proposto por Euler. Fonte: https://bit.ly/2L6j5ww

Ele nos foi apresentado no capítulo 1 como sendo a solução criada por Euler para comprovar que o problema das sete pontes de Königsberg era impossível. Existem ciclos nele? Se sim, quantos? Pare alguns segundos e tente encontrar a resposta para este enigma.

Se você encontrou os ciclos `AC`, `AB`, `ADB`, `ADC` e `BACD` assim como eu, está certo. Agora, analise o ciclo `AC` e conclua quais são os caminhos possíveis. Mais uma tarefa fácil. É possível ir do vértice `A` até o vértice `C` pelas arestas `c` e `d`. Logo, existem dois caminhos que ligam dois vértices, portanto, um ciclo. Esta situação não é a única que pode formar um ciclo, casos de vértices com arestas recursivas também contam (Even, Shimon).

É possível encontrar grafos que sejam um ciclo em sua totalidade, como um grafo em forma de colar com cinco vértices, mas em outros casos ciclos podem constituir um subgrafo, ou seja, um subconjunto dentro do conjunto maior que é o próprio grafo.

Lembre-se, matematicamente um grafo é um conjunto e, como tal, está sujeito à teoria dos conjuntos e todos os seus conceitos.

DEFINIÇÃO

Se para dois vértices distintos u e v em um grafo G existe mais de um caminho que os interliga, então o grafo G possui um ciclo (Harary, Frank; 1970).

Uma árvore é um grafo acíclico conectado. Acíclico é todo grafo que não possui ciclos, e conectado é todo grafo que para qualquer par de seus vértices existe ao menos um caminho entre eles (Bondy, J. A.; Murty, U. S. R., 2008). Logo, por razões óbvias, o grafo anterior nunca poderia ser considerado uma árvore.

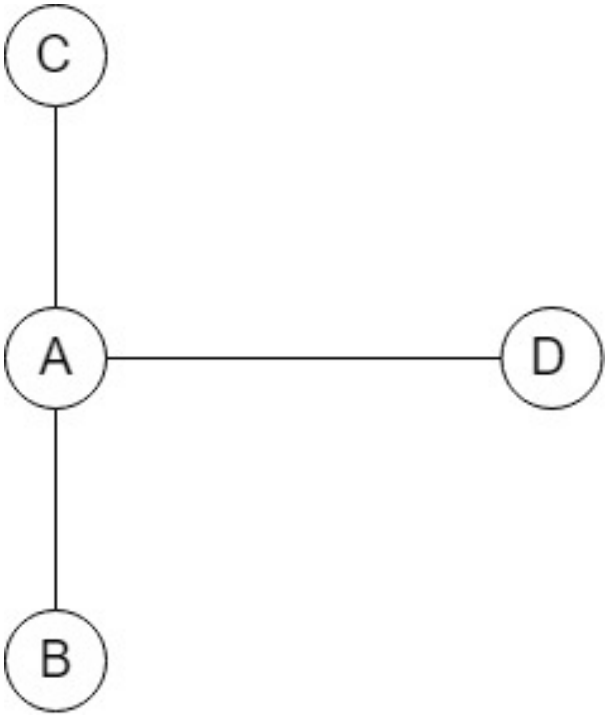


Figura 5.2: Árvore derivada do problema das sete pontes de Königsberg.

O grafo da imagem acima pode ser considerado uma árvore? Pela teoria descrita no parágrafo acima, sim, ele é uma árvore. Mas algo soa familiar? Lembra de tê-lo visto antes ou algum grafo parecido

com ele? Se sua resposta é "não", então deveria estar mais atento aos detalhes, pois a árvore acima é resultado da remoção de algumas arestas que fechavam ciclos no grafo do problema das sete pontes de Königsberg que citamos parágrafos atrás. Nela é possível sair de qualquer nó e chegar a outro trafegando nas pontes em qualquer sentido.

De forma geral, então, podemos resumir uma árvore com o quadro a seguir:

Definição

Dado um grafo G(V,E) podemos dizer que ele é uma árvore se ele for:

- Sem ciclos;
- Sem vértices com arestas recursivas;
- Conectado.

Essas são as condições básicas para que um grafo seja uma árvore. Caso uma delas seja quebrada, não existe mais uma árvore, e sim, um grafo qualquer.

Toda árvore não trivial tem no mínimo duas folhas. Folha é todo vértice que tem grau igual a 1. Já aprendemos o que é o grau de um vértice no capítulo 2 e vimos inclusive a função que a representa, `d(v)`. No exemplo da árvore de resultados da copa do mundo, a fase de grupos representaria as folhas, pois a única ligação que eles possuem são com as oitavas de final. Já as partidas as oitavas possuem grau igual a dois porque são conectados a fase de grupos e as quartas de final. Então, esta é uma verdade absoluta: toda árvore tem no mínimo dois de seus vértices com grau igual a 1 (Even, Shimon; 1979).

Note que usei o termo "não trivial", desconhecido até agora. Um grafo trivial, e consequentemente uma árvore trivial, é aquele que

possui um único vértice sem arestas (Bondy, J. A.; Murty, U. S. R., 1976). Mesmo sem arestas, um grafo trivial preenche as regras para que seja considerada uma árvore. Não possui ciclos, nem vértices com arestas recursivas e é conectado, mas fica de fora da verdade absoluta citada no parágrafo anterior.

Acreditar nesta "verdade absoluta" apresentada aqui pode ser algo realmente complicado sem uma prova. Então, pense em uma árvore gigantesca como uma que mostre a malha ferroviária de uma grande cidade, por exemplo.

Agora imagine dois vértices diferentes desta árvore conectados entre si, u e v. Se você começar a percorrer esta árvore a partir do vértice u, andando por arestas ainda não percorridas, vai acabar em um outro vértice t. O caminho feito é simples, caso contrário, um ciclo teria sido formado e você não estaria mais em uma árvore, logo $d(t)$ deve ser igual a 1. Fazendo o mesmo pelo vértice v, acabará em um outro vértice y que também terá grau igual a 1. O resto pode ser provado por indução sobre o número de vértices (Even, Shimon; 1979).

Visto que estamos em uma árvore não trivial, a quantidade de vértices n deve ser maior que 1. Se $n = 2$, teremos o caso óbvio porque somente um vértice estará ligado ao outro e cada um com grau igual a 1. Porém, se $n = 3$ e, consequentemente, $V = \{a,b,c\}$ e $E = \{ab,bc\}$, poderíamos iniciar por a e chegaríamos a c e iniciando por b chegaríamos a a, visto que a e b são diferentes, conectados entre si e o caminho feito em ambos os casos é simples. Logo, para este caso e para casos em que $n > 3$ o resultado se repetirá, toda árvore não trivial possui no mínimo duas folhas (Even, Shimon; 1979).

Note que a relação entre vértices e arestas é muito importante para as árvores. Seria possível determinar a quantidade de arestas de uma árvore de forma automática, conhecendo apenas o número de vértices? Se você é ligado aos detalhes, matou a charada, mas se ainda não percebeu acompanhe o raciocínio a seguir.

Para uma árvore T qualquer com um único vértice, temos quantas arestas? Já sabemos que uma árvore com um único vértice é uma árvore trivial e que, devido à sua trivialidade, o número de arestas é igual a 0. Até aí nenhuma surpresa, já tínhamos discutido este ponto parágrafos antes. Mas agora adicione um vértice a mais nesta árvore T, o que acontece? Como também é de nosso conhecimento, uma árvore deve ser conectada para ser considerada como tal, então automaticamente T ganha uma aresta que liga seus dois vértices. Em suma, para a árvore atual T temos a quantidade de vértices n igual a 2, e m como a quantidade de arestas igual a 1 (Even, Shimon; 1979).

Nos casos descritos acima a relação entre n e m é a mesma, a quantidade de arestas é a quantidade de vértices menos 1. Contudo, a nossa prova não pode possuir um caráter pontual, ou seja, não podemos construir uma prova para $n = 1$, outra para $n = 2$, depois outra para $n = 3$ e assim sucessivamente. Devemos construir uma única que se estenda para qualquer árvore e consequentemente para qualquer n.

Desconfiamos, mas não temos certeza ainda, que em geral, uma árvore com n vértices possui $m=n-1$ arestas. Essa hipótese bate com os casos triviais que podemos pensar, tais como uma árvore com exatamente 1 vértice (0 arestas!) ou com 2 vértices (1 aresta!). Mas como provar essa hipótese para *qualquer* árvore?

Imagino que a esta altura sua cabeça já deve ter dado um nó e você no mínimo acredita que seu cérebro derreteu. Calma, ainda há solução e ela foi desenvolvida a mais de mil anos pelos nossos amigos gregos. A indução parte do princípio de que a conclusão final contém alguma informação que não está contida nas premissas e que chegar a essa conclusão pode resultar que ela mesma seja falsa ou verdadeira (https://bit.ly/2L1k6a2).

Portanto, para provar nossa hipótese por indução, podemos assumir que ela é verdadeira para uma árvore qualquer com n vértices. Se

isso nos levar a concluir que o mesmo também é verdadeiro para qualquer árvore maior, então a hipótese está confirmada.

Pois bem, se um grafo tem n vértices, podemos temporariamente assumir que ele possui n-1 arestas. Queremos agora construir um grafo maior. Podemos fazer isso de duas formas: acrescentando um vértice ou acrescentando uma aresta.

Vamos analisar a primeira possibilidade, tentando acrescentar um vértice. Se este novo grafo tem que ser uma árvore, então não podemos deixar esse novo vértice "solto", pois isso quebraria a premissa que o novo grafo é uma árvore, pois passaria a ser não conectado. Portanto, precisamos ligar este novo vértice a algum outro vértice preexistente. Chegaríamos então a um grafo conectado com n+1 vértices e n arestas, o que confirma a nossa hipótese.

A segunda possibilidade seria aumentar o grafo acrescentando uma única aresta ao grafo. Se fizermos isso sem concomitantemente acrescentarmos outro vértice, essa aresta tem que ser criada ligando dois vértices A e B já existentes no grafo. Ora, se o grafo original era conectado pois era uma árvore, já existia um caminho entre A e B. Portanto, ao acrescentar uma nova aresta ao grafo, criou-se um novo caminho entre A e B, ou seja, criou-se um **ciclo**, e o novo grafo não pode ser uma árvore.

Como não há nenhuma forma de aumentar um grafo a não ser acrescentando vértices e arestas, podemos concluir que **qualquer** árvore com n vértices obrigatoriamente tem n-1 arestas.

Definição

Seja o grafo $G(V,E)$ conectado e acíclico, formando a árvore T; a quantidade de arestas será determinada pela quantidade de vértices menos 1, em outras palavras, $m = n - 1$.

Na Ciência da Computação, árvores são muito utilizadas pois se transformaram em estruturas que oferecem novas formas de manipulação e organização de dados. Contudo, não se confunda, mesmo sendo inseridas em um novo campo do conhecimento, elas ainda são grafos e ainda devem seguir as mesmas regras definidas tanto para grafos quanto para árvores. Contudo, em algumas ocasiões podem acabar ganhando uma nova característica, a definição de uma raiz. Também conhecida como *root*, é um vértice comum mas que se torna especial, pois uma vez que foi escolhido as arestas passam naturalmente a significar as relações de uma hierarquia entre vértices. A imagem abaixo mostra um comparativo entre duas árvores, uma sem raiz e uma com raiz.

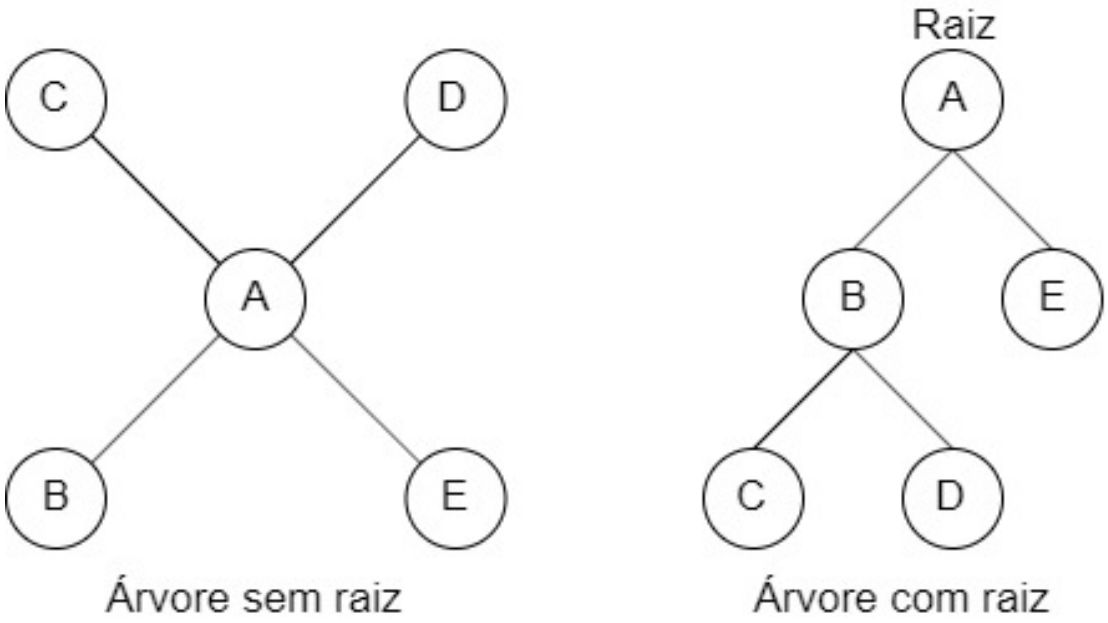


Figura 5.3: Comparação entre árvores.

Um ótimo exemplo do uso de árvores na computação é o algoritmo de Huffman utilizado para compressão de dados (Lafore, Robert; 2004). Comprimir dados é uma tarefa que acabou se tornando comum e necessária nos computadores atuais e nas nossas vidas. Quem nunca criou um arquivo `.zip` para enviar por e-mail? Hoje em dia, para nossa sorte, contamos com máquinas que possuem um bom poder de processamento e memória, e diminuir o tamanho de

um arquivo mantendo sua integridade se tornou uma tarefa, outrora difícil, fácil.

5.2 Árvores geradoras

Iniciamos esta subseção com uma pergunta. É possível transformar um grafo a tal ponto que possamos reduzir o grau de seus vértices ao menor valor possível? Diminuir o grau de um vértice implica em remover arestas conectadas a ele, conforme vimos no capítulo 2. Então a pergunta acaba perdendo seu sentido porque supostamente o grafo vai perder seu valor à medida que as arestas forem removidas. Então por que alguém faria isso?

No final do capítulo 4, vimos que você estava em apuros pois o seu chefe lhe solicitou que encontrasse uma solução para a redução de custos pedida pela diretoria. Eles queriam que você reduzisse as conexões de envio de documentos entre as filiais porque o custo estava muito alto. A sua única preocupação era manter o mínimo de conexões possíveis para que o grafo permanecesse conectado, visando não parar completamente o envio e recebimento de documentos.

Acho que já deve ter se tornado óbvio que o início da solução para o seu problema é a resposta procurada para a pergunta do primeiro parágrafo desta seção. Então, respondendo a pergunta, sim, é possível realizar essa redução, e ainda trago um novo fato à mesa: este assunto não é uma surpresa, pois já esbarramos com ele durante nossa trajetória por estes capítulos. Deixemos isto para ser discutido mais à frente, vamos pôr o nosso foco no que realmente importa.

Uma árvore geradora, ou em inglês *Spanning Tree*, é um subgrafo de um grafo G que contém todos os seus vértices e é uma árvore (Even, Shimon; 1979). A imagem abaixo esclarece esta definição.

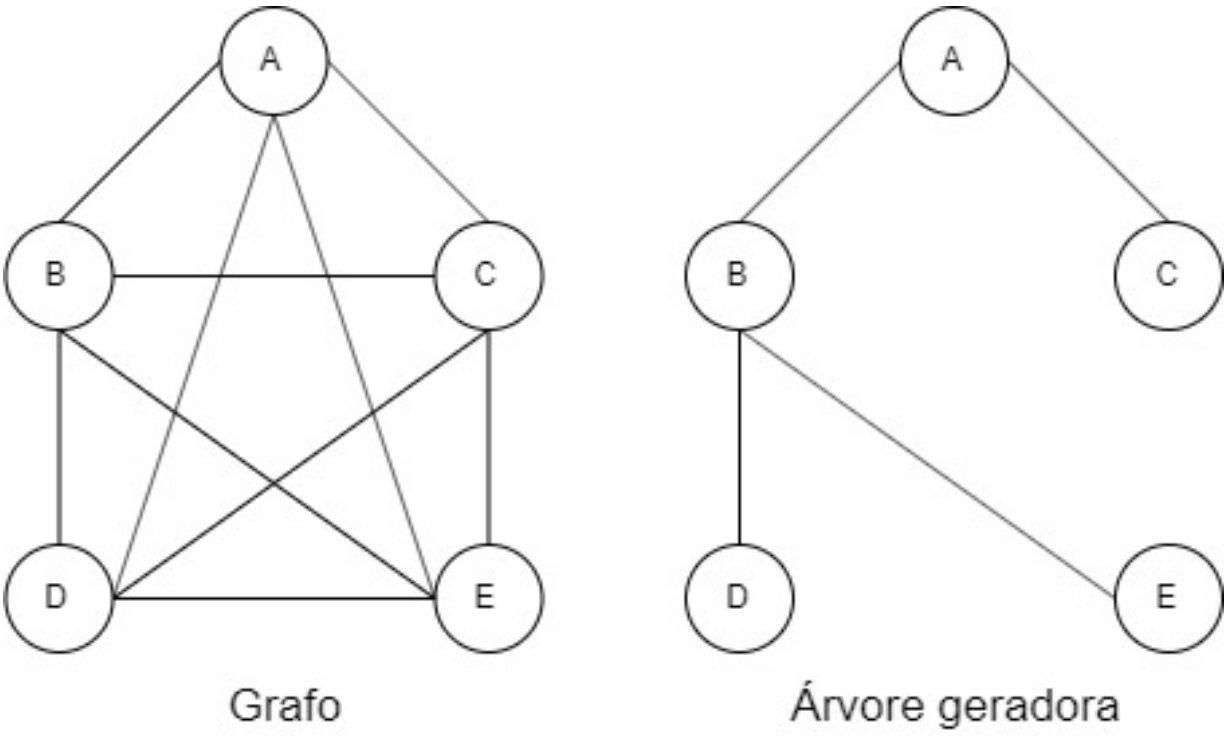


Figura 5.4: Um grafo G e sua respectiva árvore geradora.

Para o grafo `G` completamente conectado, representado na imagem, é mostrado uma árvore geradora. Contudo, diversas árvores geradoras poderiam ser definidas a partir de G pela remoção de arestas. Então, é levantado o seguinte questionamento: existe alguma ordem de importância nas arestas escolhidas para montar a árvore geradora de um grafo G? Trazendo esta dúvida para o cenário descrito pela imagem acima a pergunta se transformaria em: por que a aresta BD foi escolhida para estar na árvore geradora ao invés de CD?

Neste momento as arestas escolhidas são aquelas **necessárias** para manter o subgrafo de G uma árvore. Logo, independentemente da combinação de arestas, se elas conseguem manter conectados todos os vértices de G em seu subgrafo e tornar este subgrafo uma árvore, podem ser consideradas para uma árvore geradora.

Você percebeu que foi citado que já esbarramos com este assunto em algum momento no passado? Pois então, uma árvore geradora

pode ser derivada a partir de uma busca por profundidade porque este algoritmo (visto no capítulo 4) visita todos os nós do grafo, mas apenas uma vez. Ele nunca tenta explorar um nó já explorado pela mesma aresta percorrida ou por outra. Isto se dá devido ao esquema de marcadores usado. Caso explorasse, existiriam ou um laço ou dois caminhos para um mesmo vértice, respectivamente, gerando um ciclo e descaracterizando a árvore. Assim, o caminho construído pelo algoritmo é uma árvore geradora (Lafore, Robert; 2004). Caso você não recorde muito bem, dê uma folheada e volte ao capítulo anterior.

Existem diversos algoritmos que geram árvores geradoras. O qual veremos implementado no nosso projeto será apresentado na seção seguinte. Este, por sua vez, é uma leve variação do algoritmo de busca por profundidade e bem simples, por sinal. Mas não se engane, a geração de árvores geradoras é um assunto de grande valor e muito importante, pois como veremos à frente, a escolha da aresta a qual formará a árvore deve ser tomada considerando diversos aspectos e isso faz com que a árvore resultante do processo ganhe muita visibilidade.

5.3 Incorporando árvores geradoras à aplicação

Para que você possa atender à solicitação do seu chefe e ficar um passo mais perto daquele aumento, serão necessárias algumas pequenas mudanças no nosso projeto.

Adicione o método `arvoreGeradoraPorProfundidade()` na classe `Grafo`. Este método possuirá toda a inteligência por detrás da criação de uma árvore geradora através do método de busca por profundidade.

`src > main > java > grafo > core > Grafo.java`

```
//restante da classe
```

```
public Grafo arvoreGeradoraPorProfundidade() throws Exception {
    Grafo arvore = new Grafo();
    Stack<Vertice> roloDeBarbante = new Stack<Vertice>();
    LinkedHashSet<String> verticesVisitados = new LinkedHashSet<String>();
    List<Vertice> vertices = getVertices();

    for(Vertice v : vertices) {
        arvore.adicionarVertice(v.getRotulo());
    }

    Vertice verticePontoDePartida = vertices.get(0);
    verticesVisitados.add(verticePontoDePartida.getRotulo());
    roloDeBarbante.push(verticePontoDePartida);

    while(!roloDeBarbante.empty()){
        Vertice verticeAnalisado = roloDeBarbante.peek();
        Vertice proximoVertice = obterProximoVertice(verticeAnalisado,
verticesVisitados);
        if(proximoVertice == null){
            roloDeBarbante.pop();
        } else {
            String rotulo = proximoVertice.getRotulo();
            verticesVisitados.add(rotulo);
            roloDeBarbante.push(proximoVertice);
            arvore.conectarVertices(verticeAnalisado.getRotulo(),
proximoVertice.getRotulo());
        }
    }

    return arvore;
}
```

Se você se lembra da classe `BuscaEmProfundidade`, este algoritmo não será um desafio, mas caso não, lembre-se de que seu funcionamento se baseia na escolha de um vértice como ponto de partida, no nosso caso é o primeiro vértice da lista de `vertices`. A partir deste início, ele elege o próximo vértice que será visitado. O critério de escolha é: deve ser seu adjacente e não visitado ainda. Este processo de escolha de próximo vértice se repete para todos os vértices do grafo com algumas ressalvas para casos de vértices

já visitados em etapas anteriores e vértices que não possuem nenhuma outra adjacência, exceto aquela que fez o algoritmo chegar até ele. Resumidamente, é esse o processo. Para mais detalhes, volte algumas páginas até o capítulo 4.

O ponto de diferença entre este algoritmo e o de busca em profundidade é a montagem da árvore. Logo em seu início os vértices do grafo são adicionados na árvore recém-criada para depois no bloco `else` serem conectados. No fim de todo este processamento a árvore geradora é retornada.

`src > main > java > grafo > core > Grafo.java`

```
private Vertice obterProximoVertice(Vertice vertice, LinkedHashSet<String>
verticesVisitados){
    List<Vertice> adjacencias = getAdjacencias(vertice.getRotulo());
    for(int i=0; i < adjacencias.size(); i++){
        Vertice adjacencia = adjacencias.get(i);
        boolean naoVisitadoAinda =
!verticesVisitados.contains(adjacencia.getRotulo());
        if(naoVisitadoAinda){
            return adjacencia;
        }
    }
    return null;
}
```

Esses dois métodos, trabalhando em conjunto, tornam possível a geração de uma *Spanning Tree* pelo método de busca em profundidade. Não esqueça de adicionar os *imports* obrigatórios.

Uma vez reunido todos os elementos da nossa construção, podemos testar o que fizemos. Reutilize o código de execução do capítulo 4 e adicione a chamada ao método `arvoreGeradoraPorProfundidade(grafo)`, caso você queira realizar algum teste.

```
/*grafo feito no capítulo 4*/
/*vértices*/
```

```
/*arestas*/
/*Buscas feitas no capítulo 4*/

Grafo arvore = grafo.arvoreGeradoraPorProfundidade();
System.out.println();
System.out.println("--- Árvore geradora ---");
System.out.println("Vértices");
for(Vertice v : arvore.getVertices()) {
    System.out.println("\t" + v.getRotulo());
}
System.out.println("Arestas");
for(Vertice v : arvore.getVertices()) {
    for(Vertice adj : arvore.getAdjacencias(v.getRotulo())) {
        System.out.println("\t" + v.getRotulo() + adj.getRotulo());
    }
}
```

Espere um resultado parecido com esse.

```
--- Árvore geradora ---
Vértices
    A
    B
    C
    D
    E
    F
    G
    H
    I
Arestas
    AB
    BA
    BF
    BI
    CI
    DE
    DG
    DI
    ED
    FB
```

```
GD
HI
IB
IC
ID
IH
```

Como nosso projeto não utiliza nenhuma biblioteca para representação de grafos em forma gráfica, fica um pouco complicado desenhar uma árvore em um terminal ou *console*. Então, para que possamos entender as conexões da nossa árvore geradora, as arestas foram impressas em duplas de vértices, e a partir deste momento conseguimos ter uma ideia da árvore que foi formada. Lembre-se de que arestas redundantes foram mostradas, mas isso não significa que existam arestas paralelas.

5.4 Decisões à frente

Chegamos a um ponto decisivo na nossa jornada, em que precisamos dar um passo à frente para entrar em uma nova dimensão no mundo dos grafos. Encarar essa nova etapa da aventura é essencial para um entendimento maior sobre a teoria dos grafos e para uma abertura da nossa visão sobre as aplicações que um grafo pode ter.

Muitas vezes passamos por momentos nas nossas vidas em que precisamos fazer escolhas que são determinantes pois não há caminho de volta. Se não há um caminho de volta é porque provavelmente deve haver uma direção, na qual talvez seja possível somente ir para à frente ou para trás. E, para se fazer essa escolha é necessário pôr na balança o que para nós é o melhor. Mas, como definir o melhor? Um dos jeitos mais simples é avaliando o peso de nossas escolhas. A partir daí, conseguimos extrair a decisão que nos trará mais benefícios e consequentemente a que vai ser tomada. E isso não seria diferente para um grafo.

A fim de proporcionar uma análise mais real sobre o envio de documentos da sua empresa, você começa a pensar se existem rotas melhores ou piores que outras, o que define melhor e pior, e se é possível para um documento ser enviado de uma filial a outra em dois sentidos, ou seja, quem outrora fora o remetente se torna posteriormente o destinatário. Todas essas questões não saem da sua cabeça e você não vai conseguir ter paz enquanto não resolver esse problema.

Então, se você, jovem Skywalker, respostas procura, no próximo capítulo nos encontramos. Que a força esteja com você.

5.5 Conclusão

Uma árvore por si só já é algo fantástico, mas uma árvore geradora é um marco na teoria dos grafos e não para por aí. Analisar todas as faces deste prisma nos leva a revelações que antes poderiam ter passado despercebido, como a prova em torno da quantidade de arestas que uma árvore tem.

Obviamente, à medida que vamos adicionando mais ingredientes à mistura o resultado se torna cada vez mais interessante, como escolher quais serão as arestas que formarão a árvore geradora. Atualmente nosso único foco é extrair uma árvore a partir de um grafo desde que o grafo se mantenha conectado e acíclico.

Uma escolha entre arestas nos induz ao seguinte questionamento: qual o critério de escolha que deve ser usado. A resposta para esta pergunta se encontra no íntimo de cada grafo, em outras palavras, o que ele representa. Entender este ponto é crucial pois como visto em capítulos anteriores, arestas representam como os vértices se relacionam.

Resumo

- Se em um grafo existe mais de 1 caminho de um vértice a outro, então ele possui um ciclo.
- Dado um grafo `G(V,E)` podemos dizer que ele é uma árvore se ele for conectado e não tiver ciclos e vértices com arestas recursivas.
- Toda árvore não trivial tem no mínimo duas folhas.
- Folha é todo vértice que tem grau igual a 1.
- Um grafo trivial, e consequentemente uma árvore trivial, é aquele que possui um único vértice sem arestas.
- A quantidade de arestas de uma árvore é igual à quantidade de vértices menos 1.
- Uma árvore geradora, ou em inglês *Spanning Tree*, é uma árvore construída a partir de um grafo desde que ele permaneça conectado.
- É possível obter uma árvore geradora a partir do procedimento de busca em profundidade.

Capítulo 6
Direções e valores

Desde o momento em que nascemos, vamos acumulando experiências ao longo das quais construímos nossos sensos e valores. Para uma criança, distinguir valores pode ser uma tarefa muito difícil. Quem nunca, durante a infância, pensou no que deveria fazer? Comer mais um pedaço de chocolate ou guardar o resto na geladeira para quem não comeu? Sair ou não para brincar num dia de chuva? Assim como é difícil lidar com valores, o mesmo acontece com as direções. Será que dá para voltar por esse caminho que eu fiz? Esse caminho tem uma única direção ou é mão dupla? Questões como essas talvez irão embora quando a idade avançada nos trouxer o amadurecimento para aprender com nossos erros e acertos.

Distinguir direções e valores não é apenas um desafio para nós, seres humanos, mas também para os grafos. Veremos neste capítulo que um grafo pode ser expresso de outras formas além das quais já conhecemos, e não para por aí. Será possível combinar essas formas de representação também. Então, a partir deste ponto em particular, o problema não será mais **como** representar o grafo, visto que já o conheceremos, mas sim, avaliar o que é **melhor** ou **pior**, dado o nosso propósito. Assim como é para uma criança.

6.1 Direções

Desde o capítulo 2 viemos nos esbarrando em certos pontos aos quais ainda não demos a devida importância. Um desses pontos são os grafos orientados ou dígrafos, dos quais tanto falamos. Então chegou a hora de descobrirmos o que são, e não somente isso, mas como poderemos representá-los no nosso projeto também.

Um grafo orientado ou dígrafo segue a mesma definição de um grafo não orientado: uma estrutura composta por dois conjuntos denominados V e E, onde V representa o conjunto de vértices e E o de arestas. A única diferença é que em um dígrafo os pares de vértices são ordenados (Even, Shimon; 1979).

Um par de vértices representa a conexão, feita por uma aresta, entre dois vértices, ou seja, para o grafo não orientado imaginário `G` com `V = {A,B,C}` e `E = {e1,e2,e3,e4}`, podemos estabelecer seus pares como `[(A,B),(A,C),(B,A),(B,C),(C,A)(C,B)]`. Para estes pares não existe diferença entre AB e BA porque a aresta que conecta estes dois vértices não possui direção. Logo, é possível ir tanto de `A` para `B` quanto de `B` para `A`.

Quando é estabelecido que estes pares devem ser ordenados (seguir uma determinada ordem), isso faz com que `AB` e `BA` não sejam mais a mesma coisa. Se dizemos que `A` está ligado a `B` por uma aresta `e1`, isto significa que ela nasce em `A`, é direcionada e aponta para `B`. Assim, se levarmos em conta `BA`, a regra é a mesma, mas mudando a origem, direção e destino. Então, podemos concluir que a ordenação destes pares faz com que as arestas de um grafo ganhem direções, o que define um grafo orientado, ou, como também é conhecido, dígrafo.

A todo grafo não orientado, é possível associar um orientado, no qual cada aresta é substituída por 2 arcos de sentidos opostos e no qual não existem laços. Entenda "arco" como uma outra forma de representar uma aresta que possui uma direção (Boaventura Netto, Paulo Oswaldo; 1979). Geralmente, essa direção é representada por uma seta em uma das extremidades da aresta/arco e em alguns livros é comum o emprego deste termo para denotar arestas orientadas que tenham um formato mais encurvado.

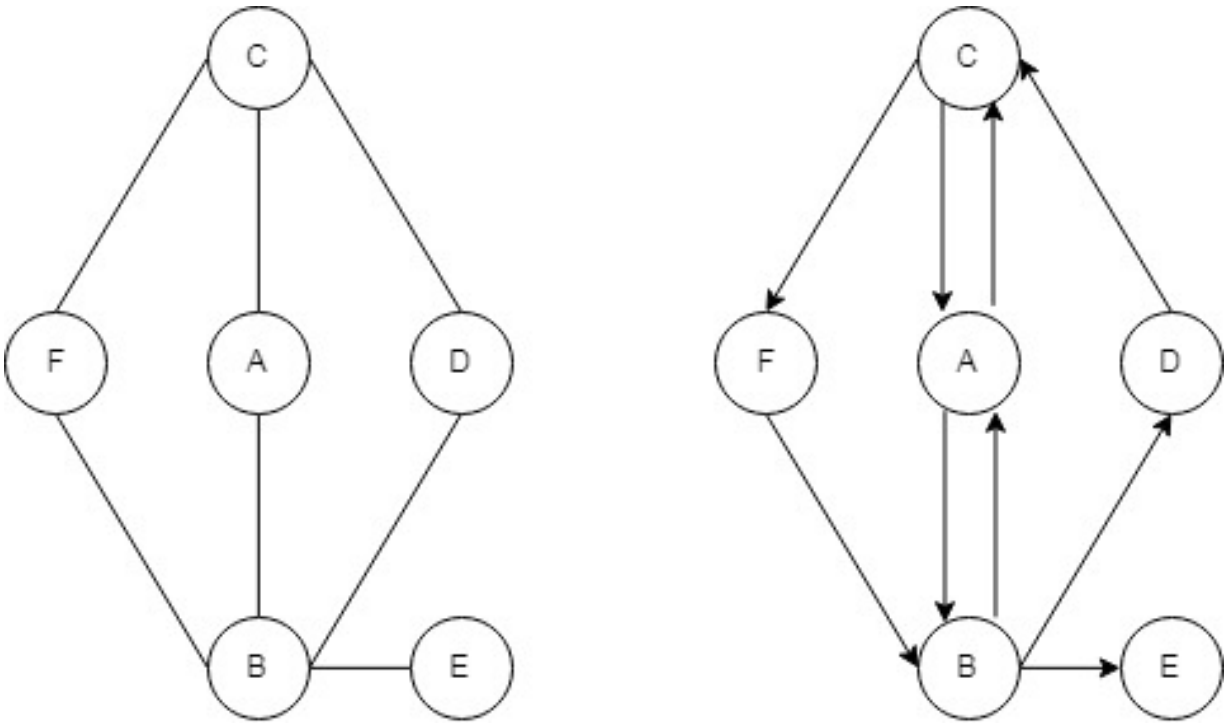


Figura 6.1: Grafo não orientado e seu respectivo dígrafo representando as ruas do meu bairro.

Imagine um aplicativo que controla e analisa o tráfego de carros em uma cidade. Neste cenário, a direção é um fator decisivo, logo o que melhor se adapta é um dígrafo porque os automóveis devem respeitar a direção da pista. Com a orientação, certos conceitos acabam ganhando mais sentido, como a adjacência e a incidência. Quando lidávamos com grafos não orientados, esses conceitos funcionavam de forma reflexiva, isto é, se um vértice A está ligado a um vértice B por uma aresta e, então um é adjacente ao outro e e é incidente a ambos. Para dígrafos, este cenário é bem diferente. Se possuirmos os mesmos A e B mas ligados por uma aresta/arco e que vai de A para B, então somente B é adjacente a A e e é incidente somente em B.

Como consequência dessas mudanças, funções como a que calcula o grau de um vértice $d(v)$ são afetadas, assim como a nossa velha conhecida matriz de adjacência. Se não se lembra mais de como

funcionam, volte aos capítulos 2 e 3 para uma relembrada. A imagem a seguir mostra o impacto dessas consequências.

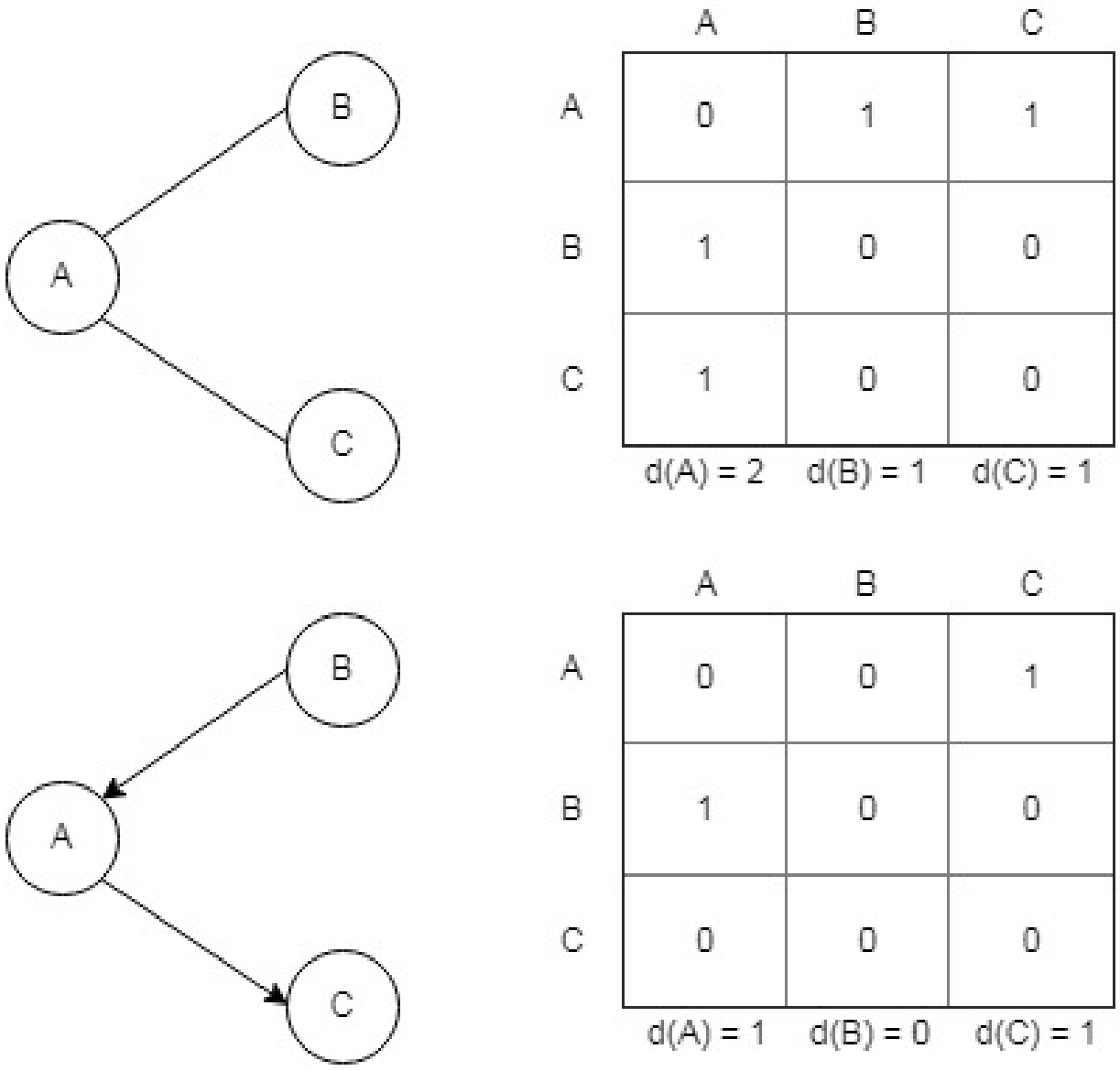


Figura 6.2: Comparativo entre grafo não orientado e dígrafo mostrando as diferenças entre suas matrizes de adjacência e funções de grau.

Note que, para a matriz do dígrafo, a terceira linha, a qual representa o vértice C , está totalmente zerada, mas em sua irmã este mesmo vértice possui uma conexão com o vértice A . Já o vértice A , que possuía duas conexões (B e C) no grafo não orientado, perde uma (B) quando analisado sob a perspectiva de

um dígrafo. Os graus desses vértices também acentuam essas mesmas diferenças, sendo que A possuía grau igual a dois e depois passou a ter grau igual a um. O mesmo acontece para o vértice B, que antes possuía grau igual a um mas, uma vez no dígrafo, passou a ter grau igual a zero. O único que se manteve em relação ao seu grau foi o vértice C.

Pode ser que, para este pequeno exemplo, você ache essas diferenças modestas, mas não se deixe levar. Uma vez que as aplicarmos em cima de algoritmos já vistos, elas tomarão uma proporção imensa. Pense em uma busca por profundidade ou na construção de uma árvore geradora, na qual as direções das arestas devem ser levadas em conta, o impacto será gigante. Mesmo assim tudo o que foi visto em capítulos anteriores continuará válido, somente serão necessárias algumas alterações no nosso projeto.

6.2 Valores

Aos 4 anos de idade, crianças já mostram vontade pelo consumo. Muitas vezes atiçadas pelos pais ou pela sua própria curiosidade elas experimentam um novo sabor em suas vidas, a vontade pela posse (Cardoso, Antônio; 2005). Com o desejo de possuir mais e mais, começamos a avaliar o valor dos nossos pertences e das nossas decisões. Eu, você, todos nós já passamos por isto. Sentimos aquela vontade de comprar algo e nos perguntamos se vale a pena. São de escolhas como essa que a vida é feita. Elas nos acompanham e acompanharão para sempre, o que muda é que com a idade o contexto das decisões deve ou deveria ser outro.

Para um grafo a realidade é a mesma. Se estamos em um vértice e temos duas arestas à frente, qual delas devemos tomar? Neste exato momento voltamos à discussão do início deste capítulo. O que é melhor ou pior? Obviamente, para cada um de nós, o que é

melhor e pior muda, alguns gostam de alguma coisa e outros não. E para um grafo, mais uma vez, a regra não poderia ser diferente. Mesmo com a mesma quantidade de vértices e arestas, e inclusive, a mesma representação gráfica, dois grafos podem ter definições de melhor e pior diferentes. A bússola que nos ajudará a encontrar a resposta para esta pergunta é o domínio tratado no grafo.

Entenda domínio como o assunto que o grafo representa, por exemplo: imaginemos um grafo que represente as linhas de energia de uma empresa de distribuição de energia elétrica. A empresa tem o desejo de reduzir custos através da otimização de suas linhas de distribuição, reduzindo conexões desnecessárias e redundantes. Logo, para este exemplo, o domínio da nossa aplicação, e consequentemente, do nosso grafo, é a representação das linhas de energia da empresa e o problema em questão é a otimização das conexões deste grafo para que sejam reduzidos custos de conexões.

Já estudamos brevemente labirintos no capítulo 4 e existe um jogo muito conhecido que se passa em um labirinto, o Pac-man. Como o blog *GameInternals - All theory, no practice* explica, o personagem principal, Pac-man, precisa comer todas as pastilhas que estão nos corredores do labirinto e, ao mesmo tempo, fugir dos quatro fantasmas (Blinky, Pinky, Inky e Clyde) que o perseguem. Em alguns pontos, existem pastilhas que são maiores e dão energia ao Pac-man, e assim ele pode os engolir e causar medo nos fantasmas, mas o efeito não é duradouro (https://bit.ly/1j22eeb).

Então, durante uma perseguição um fantasma deveria avaliar se o caminho que ele está tomando será vantajoso pois o Pac-man pode estar muito perto de uma pastilha de energia, e se ele a alcançar, o fantasma poderá ser engolido. Logo, uma decisão precisa ser tomada, arriscar para pegar o Pac-man por um caminho ou fugir por outro.

Se o fantasma pudesse atribuir notas para os caminhos, visto a situação em que ele está, quais seriam? Se o objetivo dele for

capturar o Pac-man a qualquer custo, então o caminho mais curto que leva ao Pac-man terá uma nota maior do que o caminho de saída, mas se ele for mais medroso, então o caminho de saída será o que possui a maior nota. Quanto maior a nota maior será o peso para que uma determinada escolha seja tomada.

Já vimos que um grafo pode ser usado para representar um labirinto, onde as arestas representam as alternativas de passagens entre os diversos locais. E as notas citadas acima, onde se encaixam? As notas se tornam o que conhecemos como pesos. Uma simples mudança de nome já que o seu significado se mantém o mesmo, ou seja, uma forma de avaliar o que é melhor ou pior. A única ressalva que deve ser feita é que para um grafo, uma função que gere este peso é necessária visto que ela deve avaliar o contexto no qual o grafo atua e determinar um peso para uma aresta. Um grafo que possui pesos (*weights* em inglês) em suas arestas é conhecido como grafo ponderado (Boaventura Netto, Paulo Oswaldo; 1979).

No exemplo do Pac-man, cada fantasma possui sua função específica, visto que cada um possui uma personalidade diferente, uma característica determinante. Eles se baseiam nela para tomar a decisão de qual caminho tomar. Assim, o mesmo grafo com os mesmos vértices e arestas ganha pesos completamente diferentes sob o olhar de cada fantasma. Portanto, é possível afirmar que o grande pilar que sustenta toda a ideia de um grafo ponderado é a função de geração de peso, que vai ter um papel chave na implementação deste novo tipo de grafo no projeto.

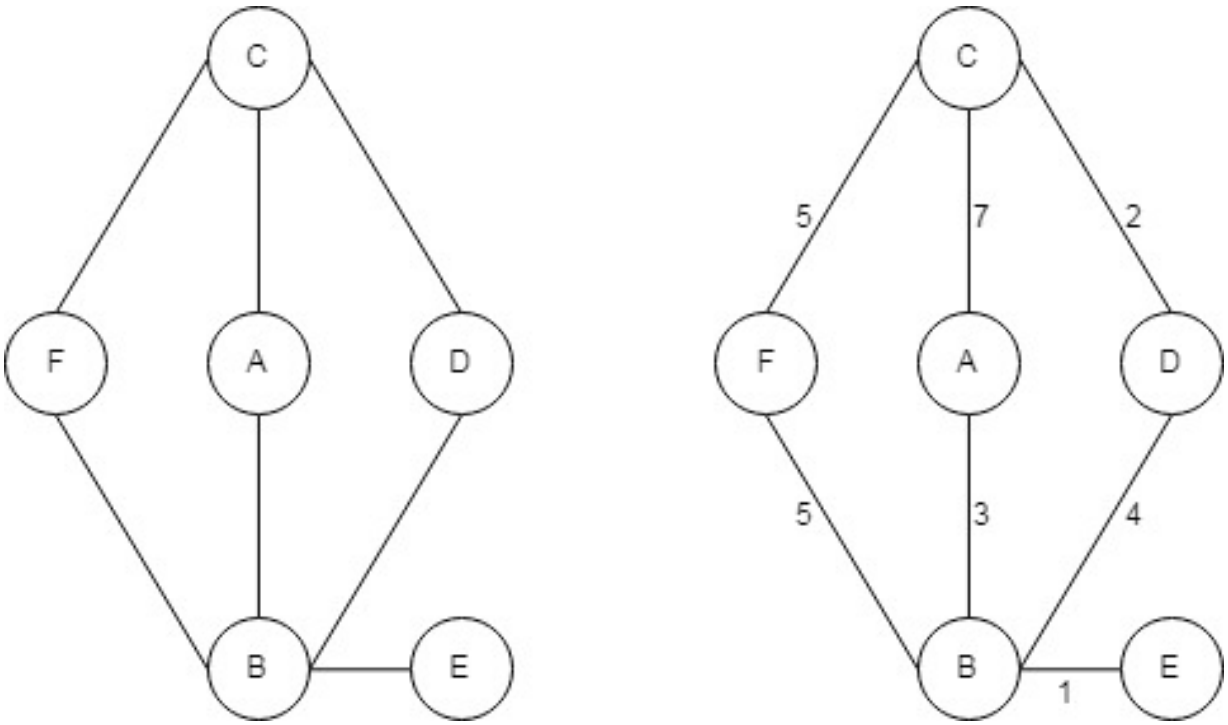


Figura 6.3: Comparativo entre grafo não ponderado e ponderado.

Representar graficamente um grafo ponderado não requer uma mudança muito brusca do que já conhecemos, basta adicionar um número a cada aresta. Caso suas arestas possuam algum tipo de rótulo, como `e1`, é possível usar a convenção **rótulo : peso**. Então a aresta de `e1` passaria a ser `e1 : 7`.

Enquanto a representação gráfica ganha uma nova característica, os conceitos já vistos nos capítulos 2 e 3 não são afetados pelos pesos. Tudo o que aprendemos sobre incidência, adjacência, graus e caminhos se mantém iguais. Buscas e árvores geradoras também não são afetadas diretamente mas formam um caso a parte porque é possível mudar suas implementações para que considerem os pesos das arestas. Não existe impedimentos em executar o que já implementamos em um grafo ponderado, somente os pesos não serão considerados.

E a direção de uma aresta é afetada pelo seu peso? Depende do segmento no qual o grafo está inserido. Caso o peso de uma aresta tenha um papel fundamental no domínio a ponto de decidir a direção

da aresta, então o grafo deverá refletir essa característica em sua representação gráfica e consequentemente em suas estruturas subjacentes. Jogos caem "como uma luva" para situações como essa. Se for montado um grafo que represente o caminho que um personagem pode fazer pelo cenário, e a direção de suas arestas forem decididas pelo grau de periculosidade que aquele caminho representa ao personagem, então podemos afirmar que, sim, a direção de uma aresta pode ser afetada pelo seu peso.

O que nos falta agora é adicionar tudo o que foi aprendido até este ponto do capítulo na nossa aplicação.

6.3 Projetando novos grafos

Para que a nossa aplicação consiga suportar grafos orientados e ponderados junto de todas as operações que realizávamos antes, precisamos realizar uma série de alterações.

Comecemos modificando nossa classe `MatrizAdjacencia` para suportar o uso de pesos em arestas e a adição de arestas direcionadas. Adicione os métodos `adicionarArestaDirecionada`, `getPeso`, `getQtdVertices`, `copiaValoresPara`, `escreveNaCelula` e altere os métodos `getAdjacencias` e `adicionarAresta`.

`src > main > java > grafo > core > MatrizAdjacencia.java`

```
//Atributos existentes acima
private Map<Integer, List<Vertice>> ancestrais;

public MatrizAdjacencia(List<Vertice> vertices){
    this.vertices = vertices;
    this.qtdVertices = vertices.size();
    matriz = new int[qtdVertices][qtdVertices];
    this.ancestrais = new HashMap<>();
    inicializarMatriz();
}
```

```
public void adicionarArestaDirecionada(int indiceVerticeInicial, int
indiceVerticeFinal, Integer peso) {
    peso = peso == null ? 1 : peso;
    Vertice verticeInicial = vertices.get(indiceVerticeInicial);
    if(indiceVerticeInicial == indiceVerticeFinal) {
        matriz[indiceVerticeInicial][indiceVerticeInicial] = peso;
        verticeInicial.addGrau();
    } else {
        matriz[indiceVerticeInicial][indiceVerticeFinal] = peso;
        Vertice verticeFinal = vertices.get(indiceVerticeFinal);
        verticeFinal.addGrau();
    }
    this.adicionarAncestral(indiceVerticeFinal, verticeInicial);
}

int getPeso(int indiceVerticeInicial, int indiceVerticeFinal) {
    return this.matriz[indiceVerticeInicial][indiceVerticeFinal];
}

public int getQtdVertices() {
    return qtdVertices;
}

void copiaValoresPara(MatrizAdjacencia matrizDestino) throws Exception {
    if(matrizDestino.getQtdVertices() < this.qtdVertices) {
        throw new Exception("Somente é possível executar cópias em
matrizes com dimensões iguais "
                + "ou a matriz de destino deve ter dimensões maiores que a
matriz de origem.");
    }
    for(int i=0; i < matriz.length; i++) {
        for(int j=0; j < matriz[i].length; j++) {
            matrizDestino.escreveNaCelula(i, j, matriz[i][j]);
        }
    }
}

private void escreveNaCelula(int linha, int coluna, int valor) {
    this.matriz[linha][coluna] = valor;
}
```

```
private void adicionarAncestral(int indiceVertice, Vertice ancestral) {
    if(this.ancestrais.get(indiceVertice) == null) {
        List<Vertice> ancestrais = new ArrayList<>();
        ancestrais.add(ancestral);
        this.ancestrais.put(indiceVertice, ancestrais);
    } else {
        this.ancestrais.get(indiceVertice).add(ancestral);
    }
}

List<Vertice> getAncestrais(int indiceVertice){
    if(this.ancestrais.get(indiceVertice) == null) {
        return Collections.emptyList();
    }
    return this.ancestrais.get(indiceVertice);
}

boolean hasAncestrais(int indiceVertice) {
    return this.ancestrais.containsKey(indiceVertice);
}
```

O método `adicionarArestaDirecionada` dá suporte à criação de arestas direcionadas entre vértices, estas podendo possuir pesos ou não. Caso uma aresta não tenha peso, o valor do parâmetro `peso` deve ser igual a `null`. Assim, o valor `1` é atribuído à célula da matriz pois simboliza uma mera conexão entre vértices. Caso seja informado um peso, o valor é armazenado na célula da matriz e desta forma é representada uma conexão entre vértices feita por uma aresta com peso.

O método `getPeso` retorna o valor da célula da matriz dado os índices dos vértices iniciais e finais. Este método pode ser invocado a partir de qualquer tipo de grafo, seja ele ponderado ou não, dígrafo ou não, ou derivado de alguma combinação. Entretanto, devemos nos lembrar que para qualquer tipo de grafo o valor zero em uma célula representa que não existe uma conexão entre vértices e que qualquer número diferente de zero representa o oposto.

O método `getQtdVertices` é um outro método do tipo *getter* o qual simplesmente retorna a quantidade de vértices presentes na matriz. Seu uso será visto mais adiante.

O método `copiaValoresPara` é usado para copiar valores de uma matriz de origem para outra de destino. A matriz de origem é representada pelo objeto que dá acesso ao método e a de destino é representada pelo parâmetro passado ao método. Parte do processo é realizado neste método e uma pequena parte no método `escreveNaCelula` que recebe como parâmetro uma combinação de linha e coluna junto com o valor que deve ser adicionado na célula correspondente.

Os métodos `adicionarAncestral`, `getAncestrais` e `hasAncestrais` não atuam diretamente no suporte a arestas orientadas e com pesos, e por tal motivo não foram relacionadas no parágrafo anterior a amostra de código. Contudo, eles são tão importantes quanto seus irmãos, pois apoiam a construção de árvores geradoras a partir de dígrafos.

Como dito, algumas alterações também tiveram de ser feitas em métodos que já existem, são elas.

`src > main > java > grafo > core > MatrizAdjacencia.java`

```
public List<Vertice> getAdjacencias(int indiceVertice) {
    int linha = indiceVertice;
    List<Vertice> adjacencias = new ArrayList<>();
    for(int j=0; j < qtdVertices; j++) {
        if(matriz[linha][j] != 0) {
            Vertice vertice = vertices.get(j);
            adjacencias.add(vertice);
        }
    }
    return adjacencias;
}

public void adicionarAresta(int indiceVerticeInicial, int
indiceVerticeFinal, Integer peso) {
```

```
    peso = peso == null ? 1 : peso;
    Vertice verticeInicial = vertices.get(indiceVerticeInicial);
    Vertice verticeFinal = vertices.get(indiceVerticeFinal);
    if(indiceVerticeInicial == indiceVerticeFinal) {
        matriz[indiceVerticeInicial][indiceVerticeInicial] = peso;
        verticeInicial.addGrau();
    } else {
        matriz[indiceVerticeInicial][indiceVerticeFinal] = peso;
        verticeInicial.addGrau();
        matriz[indiceVerticeFinal][indiceVerticeInicial] = peso;
        verticeFinal.addGrau();
    }
}
```

O método `getAdjacencias` teve que ser modificado para que dê suporte à diversidade de valores que podem existir nas células de uma matriz. Já o método `adicionarAresta` passou por modificações para o recebimento de pesos de arestas como parâmetro, assim como seu irmão `adicionarArestaDirecionada`.

Isso encerra as modificações necessárias na classe `MatrizAdjacencia`. Agora, devemos avaliar os impactos e as consequências causadas na classe `Grafo`.

Ela é de longe a mais impactada de todas pois usa diretamente os recursos expostos pela matriz de adjacência. Adapte o método `conectarVertices` para que ele aceite o uso de pesos em arestas. A mesma ressalva feita parágrafos acima em torno dos valores que este parâmetro pode receber também vale aqui. O resultado final desta modificação deve ser parecido ao código a seguir.

`src > main > java > grafo > core > Grafo.java`

```
public void conectarVertices(String rotuloVerticeInicial, String
rotuloVerticeFinal, Integer peso) throws Exception{
    if(!this.existeVertice(rotuloVerticeInicial) ||
!this.existeVertice(rotuloVerticeFinal)) {
        throw new Exception("Para adicionar uma aresta ambos os vértices
devem existir.");
    }
```

```
    criarMatrizAdjacencia();
    int indiceVerticeFinal =
this.rotulosEmIndices.get(rotuloVerticeInicial);
    int indiceVerticeInicial =
this.rotulosEmIndices.get(rotuloVerticeFinal);
    this.matrizAdjacencia.adicionarAresta(indiceVerticeInicial,
indiceVerticeFinal, peso);
}
```

Mude também o método `criarMatrizAdjacencia` para que se torne igual ao código mostrado a seguir, porque como veremos adiante ele terá um papel crucial na criação de árvores geradoras a partir de dígrafos. Note o uso, respectivamente diretamente e indiretamente, dos métodos `copiaValoresPara` e `escreveNaCelula` neste cenário.

`src > main > java > grafo > core > Grafo.java`

```
void criarMatrizAdjacencia() throws Exception {
    if(this.matrizAdjacencia == null){
        this.matrizAdjacencia = new MatrizAdjacencia(new
ArrayList<Vertice>(this.vertices));
    } else {
        int qtdVerticesNaMatriz = this.matrizAdjacencia.getQtdVertices();
        if(this.vertices.size() != qtdVerticesNaMatriz) {
            MatrizAdjacencia matrizAdjacenciaTemp = new
MatrizAdjacencia(this.vertices);
            this.matrizAdjacencia.copiaValoresPara(matrizAdjacenciaTemp);
            this.matrizAdjacencia = matrizAdjacenciaTemp;
        }
    }
}
```

Modifique o método `arvoreGeradoraPorProfundidade()` para que ele possa conectar os vértices da árvore gerada com arestas sem peso. Logo, para este caso, na chamada do método `conectarVertices` um terceiro parâmetro deve ser passado e seu valor deve ser `null` .

Como bem sabemos a classe `Grafo` encapsula o acesso à sua matriz de adjacência, e visto que adicionamos mais uma operação à

classe `MatrizAdjacencia` também devemos adicionar mais uma forma de acesso. Crie o método `getPeso`.

`src > main > java > grafo > core > Grafo.java`

```
public int getPeso(String rotuloVerticeInicial, String rotuloVerticeFinal)
{
    int indiceVerticeInicial = rotulosEmIndices.get(rotuloVerticeInicial);
    int indiceVerticeFinal = rotulosEmIndices.get(rotuloVerticeFinal);
    return matrizAdjacencia.getPeso(indiceVerticeInicial,
indiceVerticeFinal);
}
```

Alguns métodos *getters* serão necessários mais à frente. Crie os métodos `getRotulosEmIndices()` e `getMatrizAdjacencia()`.

`src > main > java > grafo > core > Grafo.java`

```
Map<String, Integer> getRotulosEmIndices(){
    return rotulosEmIndices;
}

MatrizAdjacencia getMatrizAdjacencia() {
    return matrizAdjacencia;
}
```

Este último passo encerra as modificações necessárias na classe `Grafo`. Contudo, uma dúvida paira no ar. Estaria a classe `Grafo` apta para representar um grafo direcionado? Analisando as modificações feitas até o momento podemos concluir que não. Então, o que deve ser feito para que esse objetivo possa ser alcançado?

Uma boa solução seria criar uma classe filha da classe `Grafo` chamada `Digrafo`. Devido à herança esta nova classe teria acesso a tudo que foi desenvolvido na classe `Grafo` e ainda teria espaço de adicionar novas operações que só fazem sentido no contexto de dígrafos, como a adição arestas direcionadas e uma forma personalizada de criação de árvores geradoras.

Então, sem mais delongas crie a classe `Digrafo` dentro do pacote `core` como o código adiante mostra.

`src > main > java > grafo > core > Digrafo.java`

```
package main.java.grafo.core;

public class Digrafo extends Grafo { }
```

Adicione o método `conectarVertices` a seguir.

`src > main > java > grafo > core > Digrafo.java`

```
public void conectarVertices(String rotuloVerticeInicial, String
rotuloVerticeFinal, Integer peso) throws Exception{
    if(!super.existeVertice(rotuloVerticeInicial) ||
!super.existeVertice(rotuloVerticeFinal)) {
        throw new Exception("Para adicionar uma aresta ambos os vértices
devem existir.");
    }
    Map<String, Integer> rotulosEmIndices = super.getRotulosEmIndices();
    super.criarMatrizAdjacencia();
    MatrizAdjacencia matrizAdjacencia = getMatrizAdjacencia();
    int indiceVerticeInicial = rotulosEmIndices.get(rotuloVerticeInicial);
    int indiceVerticeFinal = rotulosEmIndices.get(rotuloVerticeFinal);
    matrizAdjacencia.adicionarArestaDirecionada(indiceVerticeInicial,
indiceVerticeFinal, peso);
}
```

Muito parecido com `conectarVertices` da classe `Grafo`, este utiliza os *getters* criados parágrafos atrás para executar basicamente a mesma função, exceto pela chamada do método `adicionarArestaDirecionada` que representa o único ponto de distinção do processo. É importante citar o uso da sobrecarga de métodos aqui pois um objeto do tipo `Digrafo` não pode um oferecer um método que crie arestas que não sejam direcionadas.

Como dito anteriormente, a criação de árvores geradoras para dígrafos é um caso à parte. Então, dado este cenário, crie os

métodos `arvoreGeradoraPorProfundidade(raiz)` e `arvoreGeradoraPorProfundidade()` .

src > main > java > grafo > core > Digrafo.java

```
public Grafo arvoreGeradoraPorProfundidade() throws Exception {
    String raiz = super.getVertices().get(0).getRotulo();
    return this.arvoreGeradoraPorProfundidade(raiz);
}

public Grafo arvoreGeradoraPorProfundidade(String raiz) throws Exception {
    LinkedHashSet<String> aVisitar = new LinkedHashSet<String>();
    Digrafo arvore = new Digrafo();
    List<Vertice> vertices = super.getVertices();
    MatrizAdjacencia matrizAdjacencia = super.getMatrizAdjacencia();
    Map<String, Integer> rotulosEmIndices = super.getRotulosEmIndices();
    int indiceRaiz = rotulosEmIndices.get(raiz);

    for(Vertice v : vertices) {
        aVisitar.add(v.getRotulo());
    }

    if (raiz == null) {
        raiz = vertices.get(0).getRotulo();
    }

    aVisitar.remove(raiz);
    arvore.adicionarVertice(raiz);
    this.visitar(raiz, aVisitar, arvore);

    while(aVisitar.size() > 0) {
        if(!matrizAdjacencia.hasAncestrais(indiceRaiz)) {
            break;
        }
        String ancestral = null;
        for(Vertice a : matrizAdjacencia.getAncestrais(indiceRaiz)) {
            if(aVisitar.contains(a.getRotulo())) {
                ancestral = a.getRotulo();
                break;
            }
        }
```

```
        if(ancestral == null) {
            throw new Exception("Todos os ancestrais da raiz já foram
visitados. Dígrafo não conexo.");
        }

        aVisitar.remove(ancestral);
        arvore.adicionarVertice(ancestral);
        arvore.conectarVertices(ancestral, raiz, null);
        raiz = ancestral;
        indiceRaiz = rotulosEmIndices.get(raiz);
        this.visitar(raiz, aVisitar, arvore);
    }

    return arvore;
}
```

Devido ao uso da sobrecarga aqui também, o método `arvoreGeradoraPorProfundidade()` se faz necessário porque sem ele o método oferecido a partir de um objeto do tipo `Digrafo` seria o que foi implementado na classe `Grafo`. Tal fato poderia gerar um resultado inconsistente, pois o algoritmo desenvolvido na classe `Grafo` é voltado para grafos não direcionados.

O método sobrecarregado `arvoreGeradoraPorProfundidade()` chama o método `arvoreGeradoraPorProfundidade(raiz)`, onde `raiz` será o primeiro vértice do grafo. Logo, um dígrafo, no contexto do nosso projeto, pode cobrir os dois cenários: criação de uma árvore geradora especificando uma raiz ou não.

Olhando por um ponto de vista bem de cima, o processo descrito no método `arvoreGeradoraPorProfundidade(raiz)` não difere muito do que conhecemos. O grafo é percorrido a partir de um ponto inicial especificado pelo usuário e vai visitando de vértice em vértice utilizando a mesma lógica que uma busca por profundidade usa. Entretanto, devemos nos lembrar que um dígrafo nem sempre é conexo, e devido a tal, devemos descer um nível em nossa análise.

Caso especifiquemos uma raiz e a árvore gerada a partir dela não contenha todos os vértices que o dígrafo original tem, então

falhamos em montar a árvore geradora deste dígrafo, mas nem tudo está perdido. É possível visitar o ancestral desta raiz, caso exista, e reiniciar o processo a partir de lá. Caso não exista, o dígrafo é realmente desconexo e nenhuma árvore pode ser montada.

O registro dos ancestrais é feito no exato momento em que uma aresta direcionada é criada no grafo. Note o método `adicionarArestaDirecionada` chamando o método `adicionarAncestral`. Desta forma não inserimos complexidades quadráticas ao algoritmo da árvore geradora e podemos assim obter os ancestrais de um vértice com complexidade O(1).

Manter este registro somente faz sentido para dígrafos, pois como bem sabemos em grafos não direcionados as relações entre vértices são bidirecionais. Logo, para obter os ancestrais de um vértice basta procurar por seus adjacentes.

De uma forma bem geral, é isto que o método `arvoreGeradoraPorProfundidade(raiz)` faz. Ele marca todos os vértices do dígrafo como "não visitados" e inicia o processo a partir da raiz, caso tenha sido especificada. A raiz é adicionada à árvore, removida da marcação de "não visitada" e o método `visitar` é chamado. Caso uma raiz não tenha sido especificada é considerado que a raiz será o primeiro vértice do dígrafo.

Até aí acredito que tudo está bem, visto que esse é o processo feito na classe `Grafo` para criação de árvores geradoras. Mas o que acontece quando nem todos os vértices foram removidos da marcação "não visitado", ou seja, o tamanho de `aVisitar` for maior que 0?

É neste ponto que a análise dos ancestrais entra em ação. Se o tamanho de `aVisitar` é maior que 0, então significa que o grafo não é conexo se iniciado a partir da raiz especificada. Em outras palavras, não é possível atingir um vértice qualquer a partir da raiz, não há caminho. Logo, o processo é reiniciado a partir do ancestral

da raiz, caso exista, e esse ancestral se torna a nova raiz. Caso a situação se repita um novo ancestral da raiz é eleito.

`src > main > java > grafo > core > Digrafo.java`

```
private void visitar(String corrente, LinkedHashSet<String> aVisitar,
Digrafo arvore) throws Exception {
    for(Vertice vizinho : super.getAdjacencias(corrente)) {
        String rotulo = vizinho.getRotulo();
        if(!aVisitar.contains(rotulo)) {
            continue;
        }
        arvore.adicionarVertice(rotulo);
        arvore.conectarVertices(corrente, rotulo, null);
        aVisitar.remove(rotulo);
        visitar(rotulo, aVisitar, arvore);
    }
}
```

A mudança feita no método `criarMatrizAdjacencia` é justificada aqui já que a adição de vértices acontece no mesmo tempo em que o dígrafo é percorrido. Isso causa a recriação da matriz de adjacência, devido a suas dimensões, toda vez que um novo vértice é adicionado.

Este passo encerra as atividades necessárias nas classes do pacote `core` da aplicação.

6.4 Como usar?

Agora devemos centrar nossa atenção em como usar o que foi desenvolvido. Vamos criar um dígrafo e a partir dele montar sua árvore geradora elegendo uma raiz?

```
Digrafo digrafo = new Digrafo();

digrafo.adicionarVertice("RJ");
```

```
digrafo.adicionarVertice("SP");
digrafo.adicionarVertice("BH");
digrafo.adicionarVertice("PT");
digrafo.adicionarVertice("OS");
digrafo.adicionarVertice("SV");
digrafo.adicionarVertice("CR");
digrafo.adicionarVertice("PA");

digrafo.conectarVertices("RJ", "SP", null);
digrafo.conectarVertices("RJ", "BH", null);
digrafo.conectarVertices("RJ", "PT", null);
digrafo.conectarVertices("RJ", "PA", null);
digrafo.conectarVertices("SP", "BH", null);
digrafo.conectarVertices("SP", "OS", null);
digrafo.conectarVertices("SP", "SV", null);
digrafo.conectarVertices("SP", "CR", null);
digrafo.conectarVertices("SP", "PA", null);
digrafo.conectarVertices("SV", "PA", null);
digrafo.conectarVertices("CR", "PA", null);

Grafo arvore = digrafo.arvoreGeradoraPorProfundidade("PT");

System.out.println("--- Árvore geradora via busca por profundidade usando
raiz ---");
System.out.println();
for(Vertice v : arvore.getVertices()) {
    System.out.print("Vértice " + v.getRotulo() + " conectado a: ");
    List<Vertice> adjacencias = arvore.getAdjacencias(v.getRotulo());
    if(!adjacencias.isEmpty()) {
        for (Vertice adj : adjacencias) {
            System.out.print(adj.getRotulo() + " ");
        }
    } else {
        System.out.print("-");
    }
    System.out.println();
}
```

Espere o seguinte resultado.

```
--- Árvore geradora via busca por profundidade usando raiz ---

Vértice PT conectado a: -
Vértice RJ conectado a: PT SP
Vértice SP conectado a: BH OS SV CR
Vértice BH conectado a: -
Vértice OS conectado a: -
Vértice SV conectado a: PA
Vértice PA conectado a: -
Vértice CR conectado a: -
```

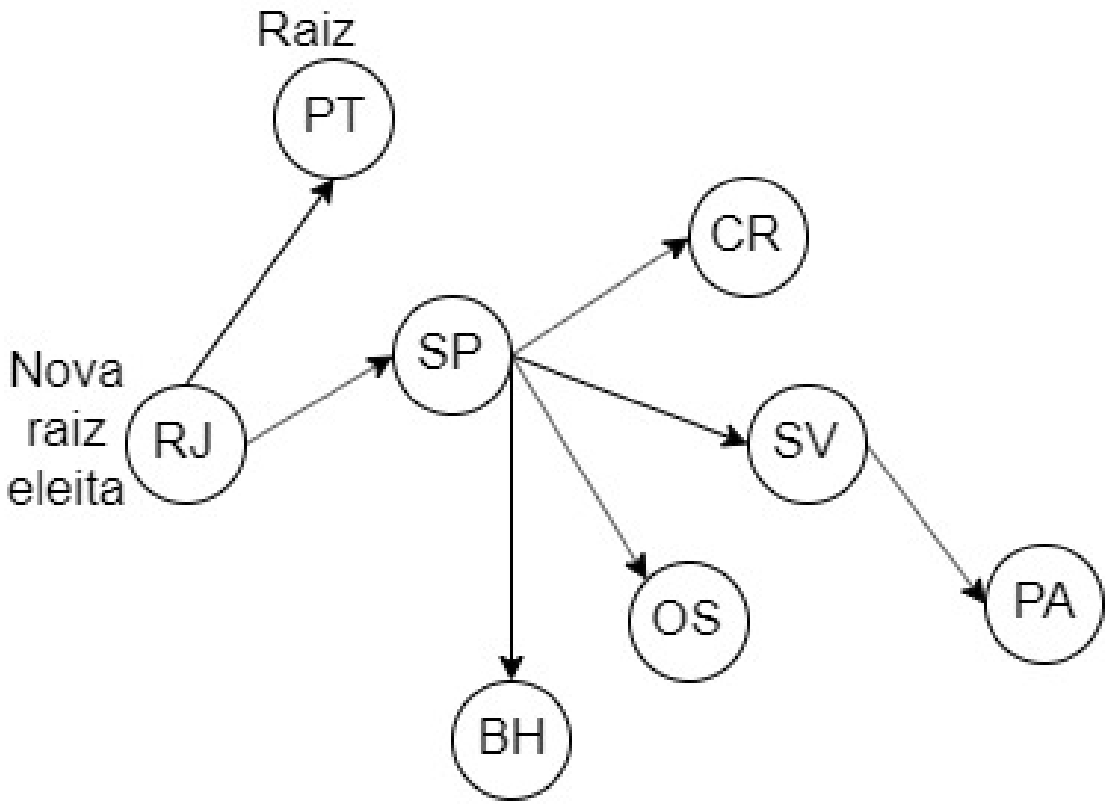


Figura 6.4: Árvore geradora criada a partir de dígrafo.

E os grafos e dígrafos ponderados ficam de fora? Claro que não!

```
Grafo grafoPonderado = new Grafo();

grafoPonderado.adicionarVertice("A");
grafoPonderado.adicionarVertice("B");
grafoPonderado.adicionarVertice("C");
grafoPonderado.adicionarVertice("D");
```

```
grafoPonderado.adicionarVertice("E");

grafoPonderado.conectarVertices("A", "B", 12);
grafoPonderado.conectarVertices("C", "E", 10);
grafoPonderado.conectarVertices("B", "D", 5);
grafoPonderado.conectarVertices("D", "A", 2);
grafoPonderado.conectarVertices("B", "E", 1);
grafoPonderado.conectarVertices("A", "C", 7);

System.out.println("Grafo Ponderado");
int peso = grafoPonderado.getPeso("A", "C");
System.out.println("Peso da aresta AC: " + peso);
peso = grafoPonderado.getPeso("B", "E");
System.out.println("Peso da aresta BE: " + peso);
```

Para o código acima espere o resultado a seguir.

```
Grafo Ponderado
Peso da aresta AC: 7
Peso da aresta BE: 1
```

Para um dígrafo ponderado eis uma amostra de como pode ser criado.

```
Digrafo digrafoPonderado = new Digrafo();

digrafoPonderado.adicionarVertice("X");
digrafoPonderado.adicionarVertice("Y");
digrafoPonderado.adicionarVertice("Z");
digrafoPonderado.adicionarVertice("W");
digrafoPonderado.adicionarVertice("V");

digrafoPonderado.conectarVertices("X", "V", 44);
digrafoPonderado.conectarVertices("Y", "W", 37);
digrafoPonderado.conectarVertices("W", "Z", 38);
digrafoPonderado.conectarVertices("X", "V", 16);
digrafoPonderado.conectarVertices("V", "X", 22);
digrafoPonderado.conectarVertices("V", "Y", 57);

System.out.println("Dígrafo Ponderado");
System.out.println("Vértices:");
```

```
for(Vertice v : digrafoPonderado.getVertices()) {
    System.out.println("\t"+v.getRotulo());
}

System.out.println();
System.out.println("Arestas:");
for(Vertice v : digrafoPonderado.getVertices()) {
    for(Vertice adj : digrafoPonderado.getAdjacencias(v.getRotulo())) {
        System.out.println("\t"+v.getRotulo()+adj.getRotulo() +
                " : " + digrafoPonderado.getPeso(v.getRotulo(),
adj.getRotulo()));
    }
}
```

Espere o resultado.

```
Dígrafo Ponderado
Vértices:
    X
    Y
    Z
    W
    V

Arestas:
    XV : 16
    YW : 37
    WZ : 38
    VX : 22
    VY : 57
```

Não se limite somente aos grafos criados nos trechos de código anteriores. Invente os seus próprios, explorando todas as possíveis combinações. Mude também a criação de vértices e arestas, adicione buscas e árvores e imprima todas as informações que quiser.

É importante relembrar que a especificação da orientação em um grafo junto da adição de pesos em arestas traz ao projeto uma

versão mais fiel do domínio. Em outras palavras, a rede de envio de malotes da sua empresa será mais bem representada assim.

Tendo isso em mente, é crucial frisar que as arestas adicionadas neste novo contexto fazem total diferença no grafo resultante porque agora elas possuem direção e peso, mostrando qual sentido um documento deve ser enviado.

Estamos mais próximos do fim desta jornada e cada vez mais conseguimos produzir uma representação mais fiel do nosso problema real. Este é o verdadeiro desafio aqui e para superá-lo devemos pensar na construção de ferramentas que operarão sob esta nova perspectiva.

Como determinar o menor custo de uma filial até outra? Quais caminhos eliminar para uma redução total de custos? Para responder estas e muitas outras perguntas que virão precisaremos de novas ferramentas que estejam à altura desses novos problemas. Assim como um canivete suíço, que consegue reunir vários apetrechos em um lugar só e salvar o seu dia, será esse kit que desenvolveremos. Ele será a conclusão do nosso projeto e a sua porta de entrada para um mundo de possibilidades.

6.5 Conclusão

Direções e pesos para arestas não são meras características. Elas levam nossos grafos para abstrações que estão mais perto da realidade dos que já montamos em capítulos anteriores. Contudo, não pense que grafos não orientados não têm o seu valor. Cada tipo de grafo tem um motivo certo para ser usado e o que vai decidir isso será o problema tratado.

Forçar uma abordagem erroneamente nos levará a um grafo que não refletirá a nossa necessidade e consequentemente não trará os resultados esperados. Tenha sempre isso em mente.

Outro ponto importante é que este capítulo nos oferece novas possibilidades e um maior entendimento sobre o mundo à nossa volta. As relações que antes eram mútuas agora não são mais e isso faz com que conceitos outrora estudados tenham mais sentido agora, como a incidência e adjacência. O desabrochar de novas possibilidades acontece como reflexo da incorporação da direção e peso em uma aresta, trazendo novas ferramentas e técnicas que terão uma maior aplicabilidade para questões do nosso cotidiano.

Resumo

- Um grafo orientado ou dígrafo segue a mesma definição que um grafo não orientado possui, ou seja, uma estrutura composta por dois conjuntos denominados V e E, onde V representa o conjunto de vértices e E o de arestas. Contudo, a única diferença é que em um dígrafo os pares de vértices são ordenados.
- Um par de vértices ordenado faz com que as conexões AB e BA não sejam mais a mesma coisa. Se a conexão é denotada por AB então é porque existe uma aresta direcionada que nasce em A e vai para B.
- É possível extrair de todo grafo não orientado um orientado. Cada aresta deve ser substituída por 2 arcos de sentidos opostos e não podem existir laços.
- Arco é a representação de uma aresta com uma seta cujo sentido corresponde à orientação do par ordenado.
- Para grafos orientados alguns conceitos ganham mais sentido, como a adjacência e a incidência. E, por tal motivo, funções como a que calcula o grau de um vértice e a matriz de adjacência são afetadas.
- Um grafo que possui pesos em suas arestas é conhecido como grafo ponderado.

Capítulo 7
O canivete suíço

Um canivete representa a reunião de vários utensílios que podem literalmente "quebrar um galho" nas horas de aperto. É geralmente pequeno, cabe em uma mão e é fácil de levar para todo lugar. É ao mesmo tempo uma ideia: a de juntar tudo o que precisamos em um lugar só e de ser algo prático. Exatamente o que queremos neste momento.

O nosso canivete não terá lâminas, chaves de diversos tipos ou serras, e nem será suíço. Mas será composto de algoritmos que darão um impulso maior ao projeto. Representarão a implementação de diversas técnicas que aprenderemos no decorrer deste capítulo, para que no final tenhamos claro em nossas mentes que este deve ser o nosso principal objetivo: tornar o canivete maior a qualquer custo. Quanto mais ferramentas adicionarmos a ele, mais resultado entregaremos para o usuário final.

Estamos na penúltima etapa da nossa jornada e já temos toda a base necessária para ultrapassar a barreira que nos impede de alçar voos maiores, em outras palavras, ir para o lugar onde estão os verdadeiros desafios.

7.1 O algoritmo de Dijkstra

Criado por Edsger Dijkstra em 1959 (Lafore, Robert; 2004), este algoritmo é um dos mais conhecidos entre seus irmãos porque nos dá a solução para um problema corriqueiro. Qual é o melhor caminho entre dois pontos?

Sempre me fazia essa pergunta quando ia ao shopping com minha avó porque, como toda boa avó, ela gostava de andar por todos os corredores do shopping e olhar todas as vitrines. Eu ficava tentando entender qual era o motivo por trás dessa andança porque sabia que só tínhamos que ir a uma loja específica e depois ir embora. Anos depois, resolvi perguntar a ela a razão pela qual dávamos "voltas ao mundo", para, no fim, saber que bastava ter virado à esquerda e depois à direita.

Ela respondeu que não sabia qual era o melhor caminho, então ela ficava dando voltinhas no shopping e uma hora ela acabaria encontrando a loja. É claro que quando se é uma criança fazer isso é uma tortura, mas hoje em dia penso que este algoritmo podia ter me salvado de boas andanças. Para a minha sorte todo fim de passeio vinha acompanhado de um sorvete, então valia a pena andar tanto.

O algoritmo consegue determinar, a partir de um vértice inicial em um grafo finito cujas arestas possuem pesos positivos, quais são os melhores caminhos para todos os outros vértices, caso não seja especificado um vértice de destino (Even, Shimon; 1979). Quando digo "melhor" ou "melhores" me refiro a algo que faça sentido para você porque, como já vínhamos discutindo capítulos atrás, esse conceito muda de acordo com o contexto em que ele está. No meu caso (exemplo da minha vó) o "melhor" seria um caminho mais curto, isto é, um que possuísse a menor quantidade em metros. Isso fará mais sentido adiante, quando precisarmos construir a parte do algoritmo que define o que é "melhor" para o grafo.

Conceitualmente, sua ideia consiste em manter dois conjuntos de vértices, T e P. O conjunto T é formado por vértices temporariamente marcados, isto é, seu menor caminho foi calculado de forma não definitiva, ele ainda pode mudar se for encontrada uma aresta que resulte em um caminho com menor custo. E o conjunto P representa vértices que foram permanentemente marcados, ou seja, aqueles que já tiveram seu menor caminho calculado de forma definitiva. A sustentação destes dois conjuntos

forma o núcleo do algoritmo, cujo resultado final será T vazio, pois todos os vértices foram processados, e P com todos os vértices com seus menores caminhos calculados (Even, Shimon; 1979).

O grafo começa a ser percorrido a partir do vértice sinalizado como de origem em direção aos seus adjacentes. Um a um são contabilizados temporariamente os custos dos trajetos do vértice atual, nesse caso o de origem, até seus adjacentes. Entenda por custo o valor do peso da aresta.

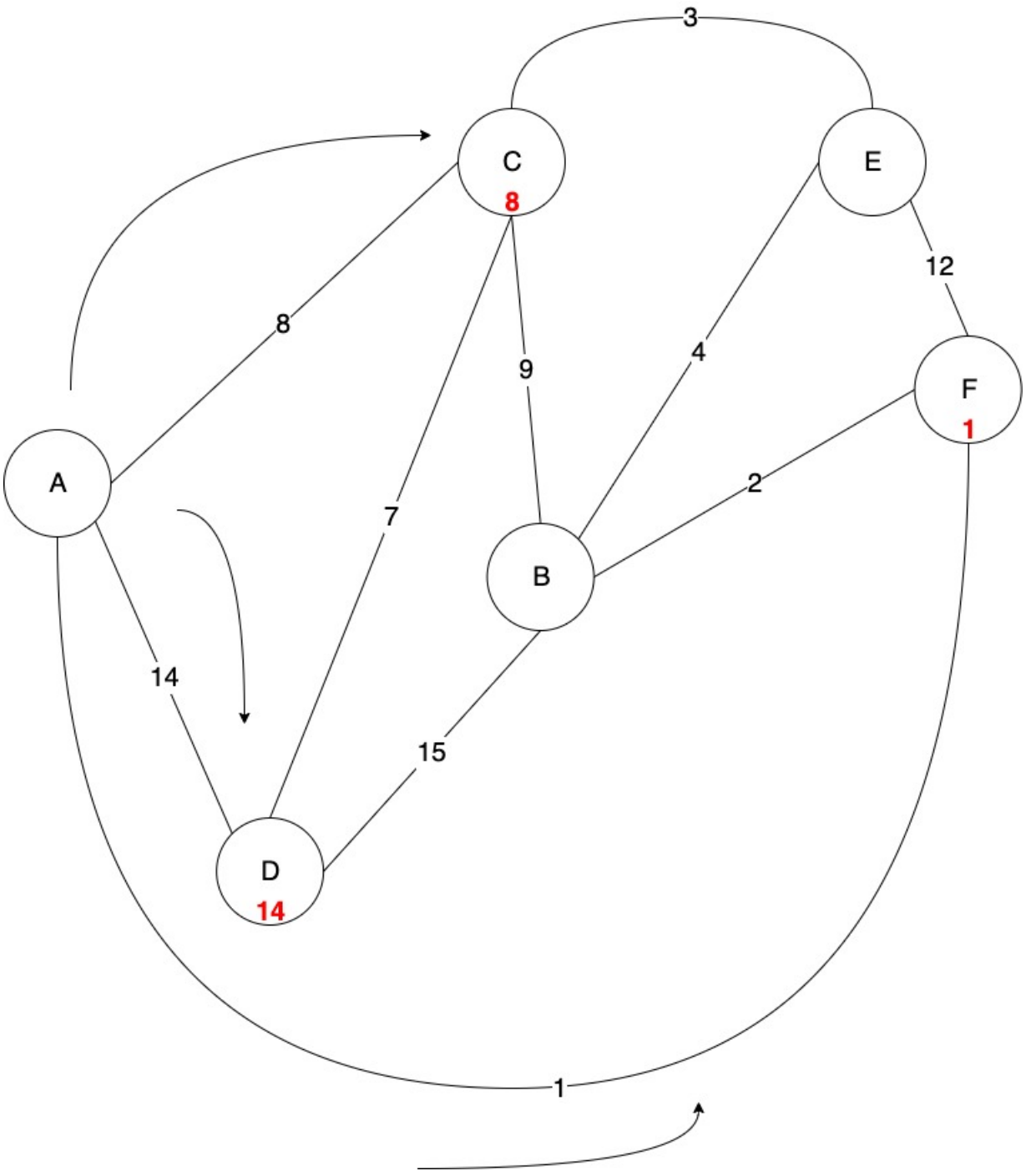


Figura 7.1: Funcionamento do Algoritmo de Dijkstra

Após contabilizados, é escolhido o primeiro adjacente de menor custo, ele se torna o atual e o vértice que guiou o trajeto até ele é posto fora do universo observável do algoritmo. O processo se repete, são contabilizados temporariamente os custos do vértice

atual, agora adjacente ao de origem, até seus adjacentes, sendo que eles devem estar dentro do universo observável do algoritmo. O custo de cada adjacente é calculado somando o custo do vértice atual com o peso da aresta incidente.

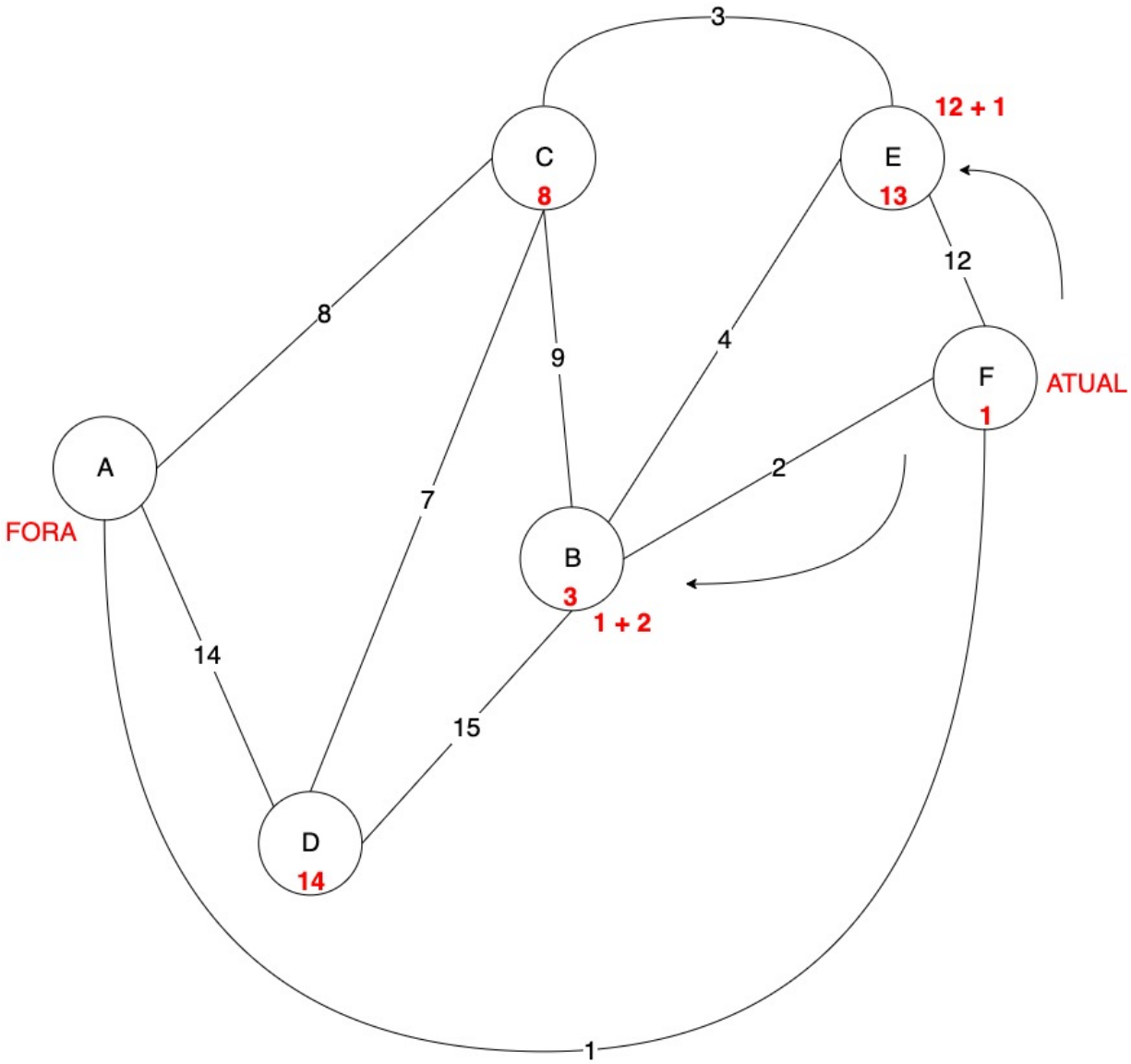


Figura 7.2: Funcionamento do Algoritmo de Dijkstra

Todo este processo vai se repetindo para o grafo inteiro até o ponto em que um vértice com custo já calculado é revisitado (vértices C, D e E). Neste momento é comparado se o custo do trajeto atual é menor que o custo já calculado. Caso seja, o custo do vértice é

atualizado, caso não, o grafo continua a ser percorrido. Isso é o que acontece com o vértice E. O vértice B é retirado do universo observável.

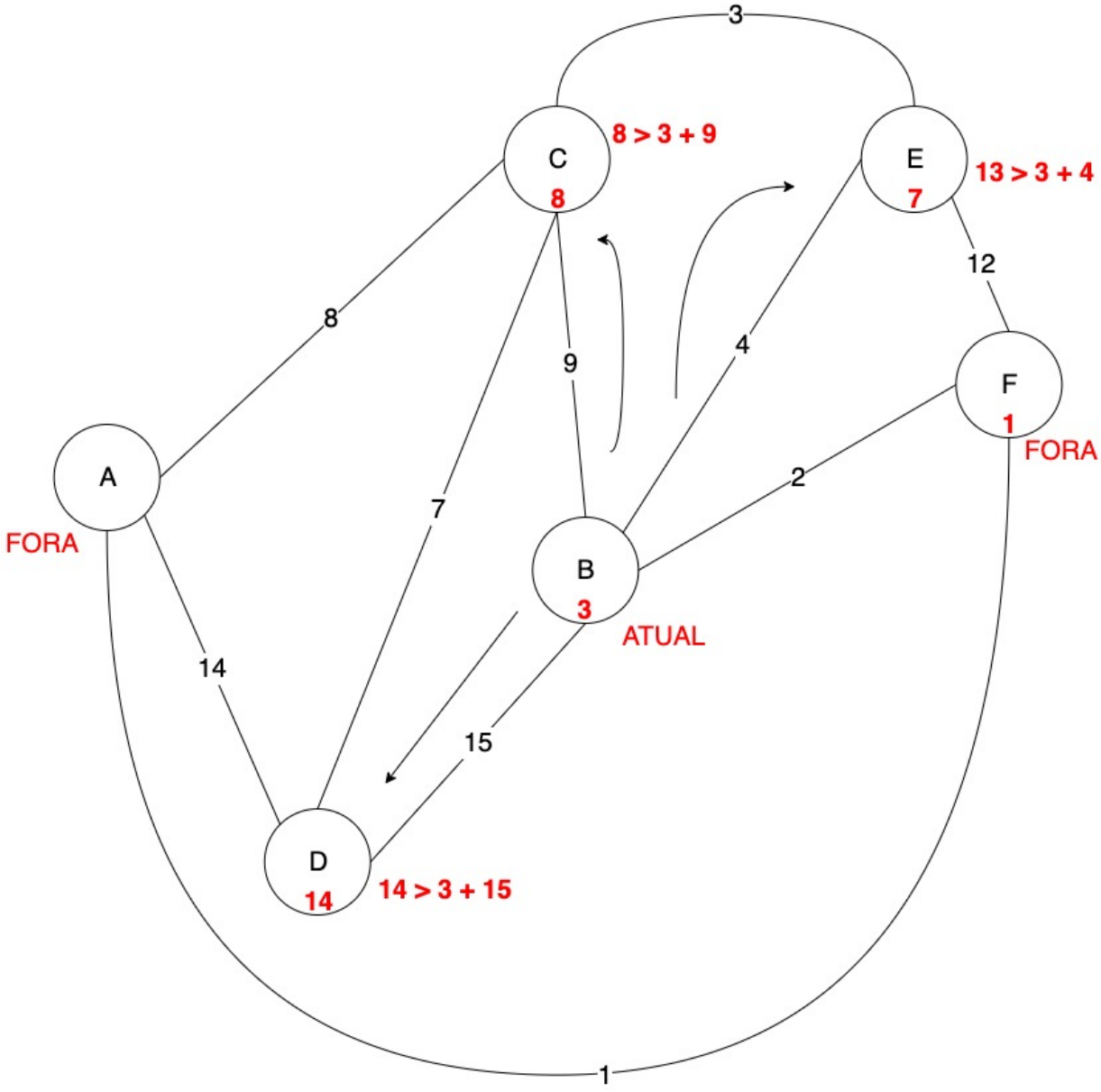


Figura 7.3: Funcionamento do Algoritmo de Dijkstra

O processo se repete mais uma vez. O vértice E é o atual e o único vértice visitável é C. A comparação de custos acontece e é concluído que o custo original do vértice é menor do que o trajeto

corrente, então se mantém o custo original. O vértice E é retirado do universo observável.

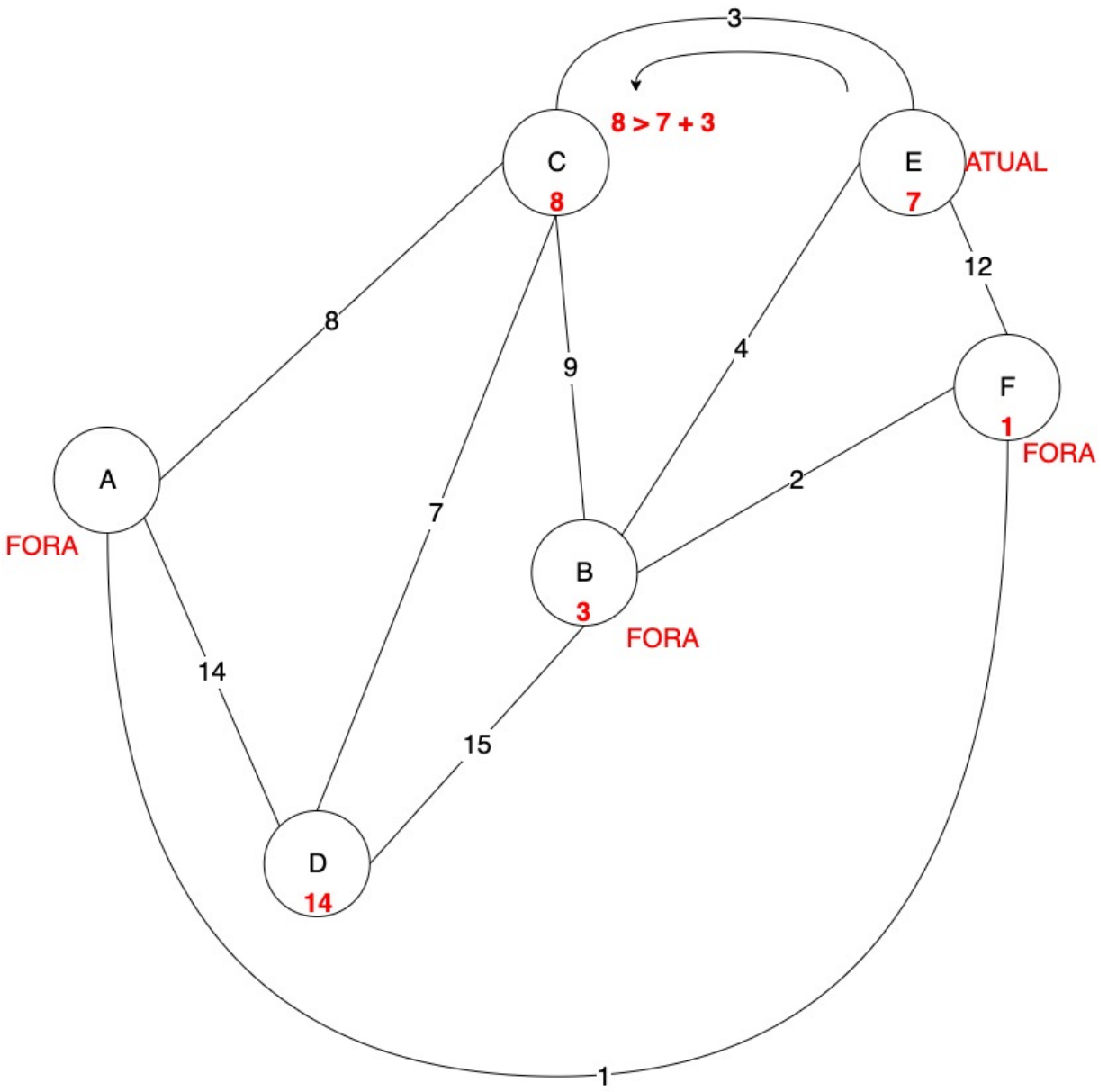


Figura 7.4: Funcionamento do Algoritmo de Dijkstra

O vértice C se torna o atual e sua única conexão passível de exploração é com o vértice D, já que os vértices A, B e E estão fora. A verificação de custos é feita e o custo original de D é mantido. C é removido e o vértice D se torna o atual.

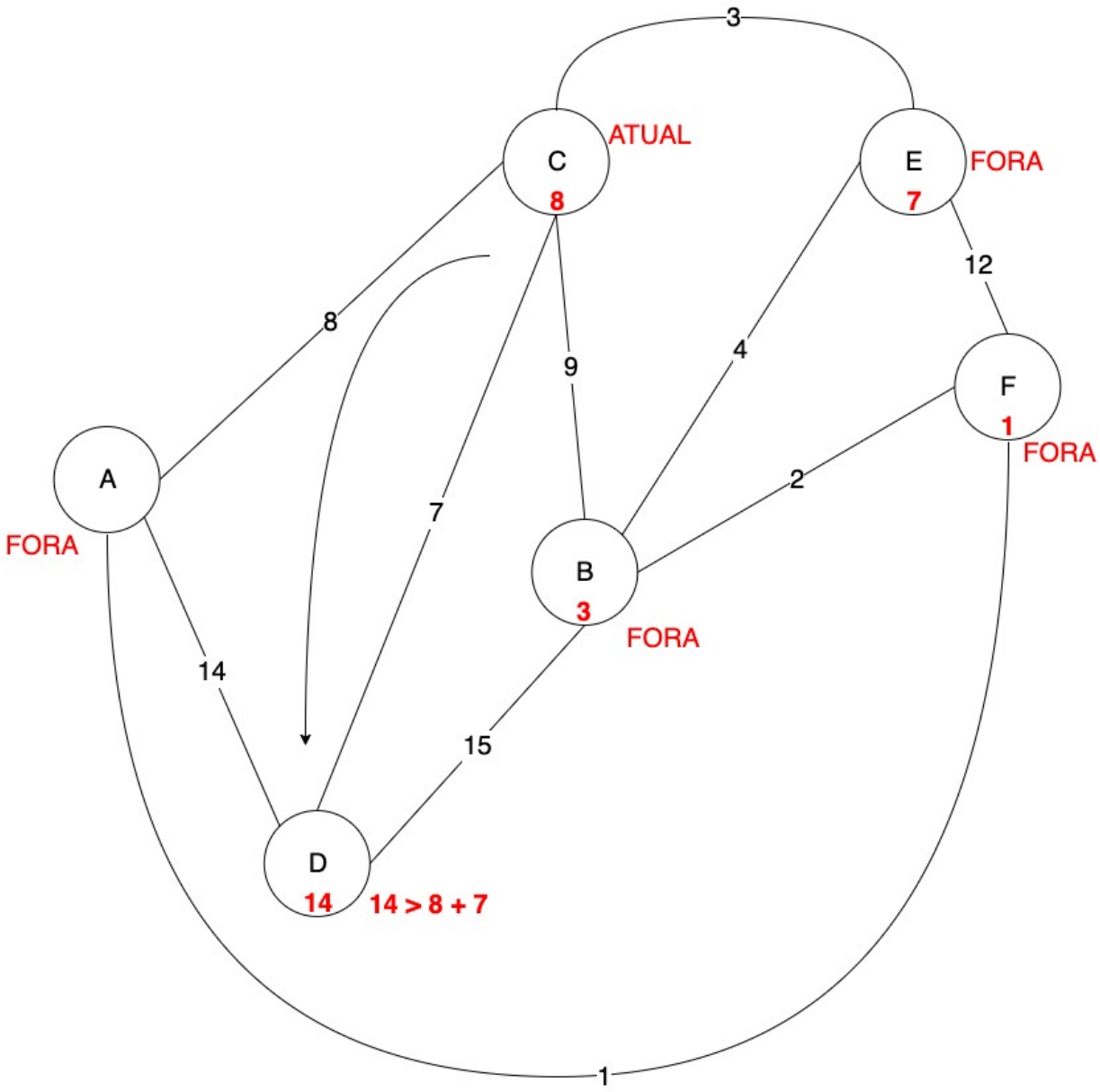


Figura 7.5: Funcionamento do Algoritmo de Dijkstra

Com o vértice D como atual, para onde ir? Todos os vértices A, B e C já não estão mais dentro das possibilidades de visita do algoritmo, então o que fazer? Simples! Em casos como este o algoritmo chega ao fim.

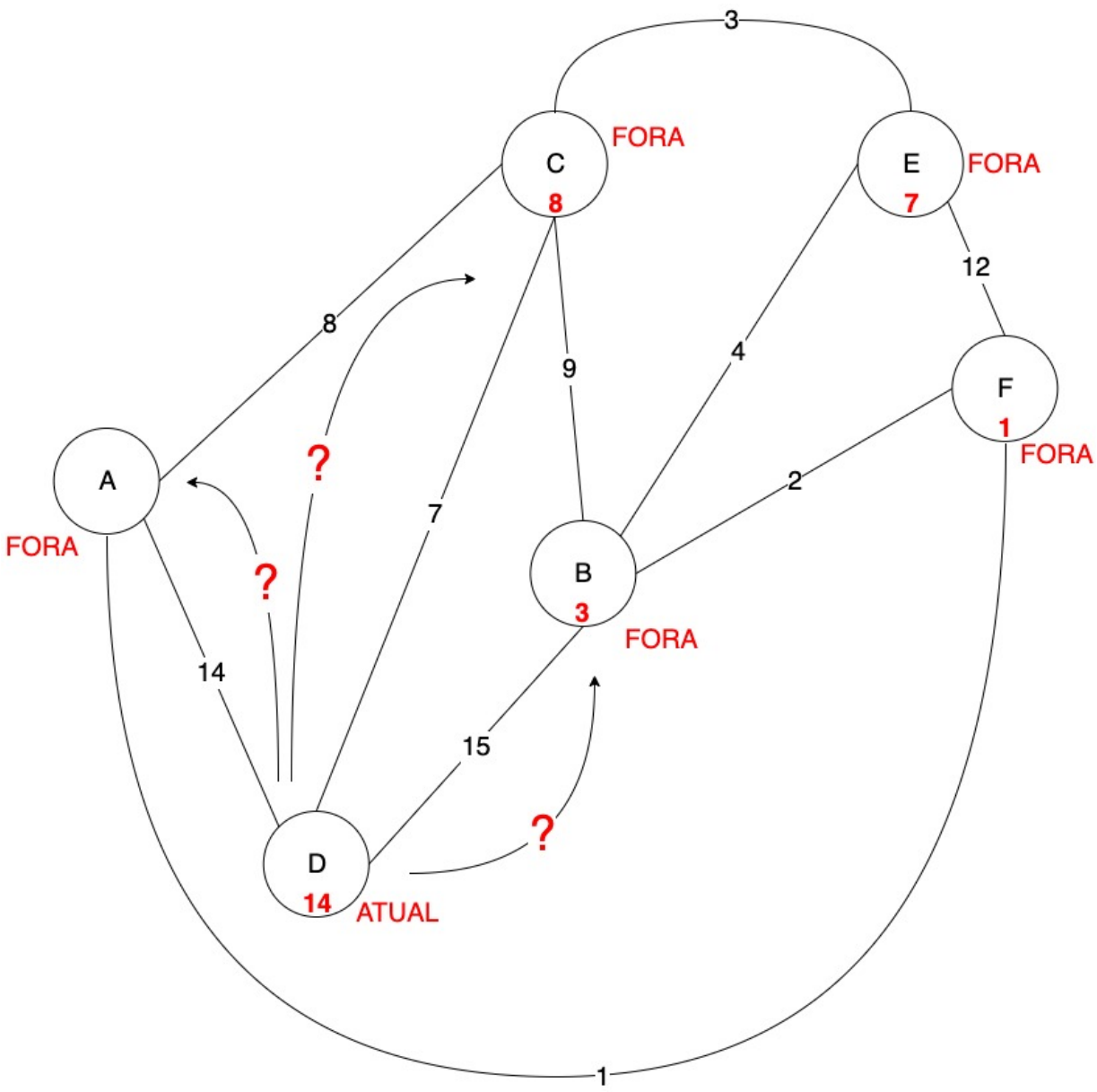


Figura 7.6: Funcionamento do Algoritmo de Dijkstra

Pondo em prática

Para que possamos implementá-lo no nosso projeto, crie a classe `AlgoritmoDijkstra` dentro do pacote `util`. Este pacote englobará os algoritmos utilitários que criaremos no decorrer deste capítulo.

`src > main > java > grafo > util > AlgoritmoDijkstra.java`

```java
package main.java.grafo.util;

public class AlgoritmoDijkstra {

}
```

Crie o método `processar(origem, destino, grafo)`. Nele será desenvolvida a implementação do algoritmo de Dijkstra, onde será possível especificar uma origem, um destino (opcional) e o grafo ou dígrafo no qual o algoritmo deve ser executado.

`src > main > java > grafo > util > AlgoritmoDijkstra.java`

```java
package main.java.grafo.util;

public class AlgoritmoDijkstra {
    public Map<String, Info> processar(String origem, String destino,
Grafo grafo) {
        try {
            //verifico se ambos os vértices de origem e destino existem.
Caso não existam, uma exceção é propagada.
            grafo.getVertice(origem);
            grafo.getVertice(destino);
        } catch (Exception e) {
            throw e;
        }

        Map<String, Info> infoVertice = new HashMap<>();
        infoVertice.put(origem, new Info(0, null));

        Set<String> aVisitar = new HashSet<>();
        aVisitar.add(origem);

        while (aVisitar.size() > 0) {
            String melhorVertice = null;
            int menorDistancia = Integer.MAX_VALUE;
            for (String v : aVisitar) {
                Info info = infoVertice.get(v);
                if (info.distancia < menorDistancia) {
                    melhorVertice = v;
                    menorDistancia = info.distancia;
```

```
                }
            }

            if (melhorVertice.equals(destino))
                break;

            aVisitar.remove(melhorVertice);

            for(Vertice vizinho : grafo.getAdjacencias(melhorVertice)) {
                String rotulo = vizinho.getRotulo();
                int distancia = menorDistancia +
grafo.getPeso(melhorVertice, rotulo);
                if(infoVertice.containsKey(rotulo)) {
                    Info info = infoVertice.get(rotulo);
                    if(distancia < info.distancia) {
                        info.distancia = distancia;
                        info.predecessor =
grafo.getVertice(melhorVertice);
                    }
                } else {
                    infoVertice.put(rotulo, new Info(distancia,
grafo.getVertice(melhorVertice)));
                    aVisitar.add(rotulo);
                }
            }
        }

        return infoVertice;
    }
}
```

A implementação é dividida em três partes: checagem, inteligência e retorno. A primeira etapa, compreendida no bloco try-catch do início do método, verifica se ambos os vértices de origem e destino existem no grafo passado. Lembre-se de que o vértice de destino é opcional, logo ele só pode ser verificado caso tenha sido informado. A segunda parte, da declaração da variável `infoVertice` até o fim do bloco `while`, é a que concentra a inteligência do algoritmo de Dijkstra e é onde concentraremos nossos esforços. A última parte, a instrução `return`, é o retorno dos dados gerados pelo algoritmo.

A etapa de inteligência começa com a adição do vértice de origem na estrutura (`infoVertice`) que mantém os custos temporários e permanentes, a qual é representada por um *map*, cuja chave é o rótulo de um vértice (nesse caso, o vértice de origem), e seu valor é composto pela informação sobre seu custo associado e o vértice que o precede. Feito isso, o vértice de origem também é adicionado à estrutura (`aVisitar`) que contabiliza os vértices que devem ser visitados. É iniciado então um processo iterativo que só chega ao fim quando não existir mais nenhum vértice em `aVisitar` .

Este processo determina qual o melhor vértice entre os presentes em `aVisitar` , e uma vez determinado verifica se o vértice informado como destino é o mesmo determinado como melhor vértice. Caso sejam, o algoritmo chega ao fim e as informações computadas até o momento em `infoVertice` são retornadas. Caso contrário, o vértice escolhido como melhor é removido de `aVisitar` , ou seja, ele já foi visitado, e é iniciado um subprocesso de avaliação dos vértices adjacentes ao melhor. Para cada adjacente é calculada sua distância, que se resume à soma da distância do melhor vértice com o peso de sua aresta incidente. Encontrada sua distância, é verificado se este vértice adjacente já se encontra em `infoVertice` . Caso não exista (bloco `else`), o vértice adjacente é adicionado em `infoVertice` e marcado como visitado.

Caso exista, é verificado se a distância calculada é menor que a distância temporária presente em `infoVertice` , pois, caso seja, será necessário atualizar a distância e o vértice precedente deste adjacente. Essa atualização é necessária porque representa que um trajeto com custo menor através de um outro vértice precedente foi encontrado. Note o uso da classe `Info` na implementação do método `processar` , ela é uma classe interna da classe `AlgoritmoDijkstra` .

`src > main > java > grafo > util > AlgoritmoDijkstra.java`

```
package main.java.grafo.util;
```

```java
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Vertice;

public class AlgoritmoDijkstra {

    private static AlgoritmoDijkstra algoritmoDijkstra;

    private AlgoritmoDijkstra(){}

    public static AlgoritmoDijkstra getInstance(){
        if(algoritmoDijkstra == null){
            return new AlgoritmoDijkstra();
        }
        return algoritmoDijkstra;
    }

    public Map<String, Info> processar(String origem, String destino,
Grafo grafo) {
        //implementação do método processar
    }

    public class Info {
        public int distancia;
        public Vertice predecessor;

        Info(int distancia, Vertice predecessor) {
            this.distancia = distancia;
            this.predecessor = predecessor;
        }
    }
}
```

O algoritmo de Dijkstra pode ser chamado em qualquer lugar da seguinte forma:

```
Grafo grafo = new Grafo();

//adição de vértices

//criação de arestas com peso

Map<String, AlgoritmoDijkstra.Info> menoresCaminhos =
    AlgoritmoDijkstra.getInstance().processar("X", "Y", grafo);
Set<String> keys = menoresCaminhos.keySet();
for(String key : keys) {
    AlgoritmoDijkstra.Info info = menoresCaminhos.get(key);
    String predecessor = info.predecessor == null ?
            ""
        :
            info.predecessor.getRotulo();
    System.out.println(
        key + " : " + info.distancia + " - " + predecessor);
}
```

7.2 O algoritmo de Floyd-Warshall

Na seção anterior, vimos que o algoritmo de Dijkstra se propõe a encontrar o caminho mais curto entre dois pontos, desde que exista um caminho que leve de um a outro. Contudo, ampliemos um pouco este universo. E se quiséssemos encontrar todos os caminhos mais curtos entre todos os pares de vértices do nosso grafo, seria possível? Evidentemente, pois você pensa que basta aplicar o algoritmo de Dijkstra para todos os pares possíveis e, pronto, o problema está resolvido.

Devo concordar com você, caro leitor. Sim, é possível resolver o problema do caminho mais curto entre todos os pares desta forma, mas lhe faço uma outra pergunta em seguida. Seria esta a forma mais eficaz? É isso que veremos agora.

Em 1962, Robert Floyd e Stephen Warshall publicaram um algoritmo que resolveria este problema de uma forma mais eficaz que a aplicação repetida de Dijkstra. Tal algoritmo terminou por levar o nome de ambos e funciona da seguinte forma (Lafore, Robert; 2004).

Entendendo o funcionamento

Cada vértice do grafo deve ser analisado categoricamente. Após recuperado para análise, todas as conexões diretas **do grafo** são recuperadas e analisadas uma a uma. Quando se usa o termo "conexão direta", quer se dar a ideia de um vértice ligado a outro por uma aresta - essa escolha de nome é justificada mais à frente. Uma vez recuperada é obtida sistematicamente uma conexão alternativa que envolva os vértices da conexão direta, sendo intermediados pelo vértice recuperado no início do parágrafo.

Considere um grafo G(V,E) que possui os vértices `V` e `U` ligados por uma aresta `e`, formando uma conexão direta. Em seguida, temos `X`, um outro vértice de G. Uma conexão alternativa neste contexto seria formada pelo caminho `P: V-e1-X-e2-U`, onde o vértice `X` representa o vértice intermediário da conexão alternativa `V -> U`.

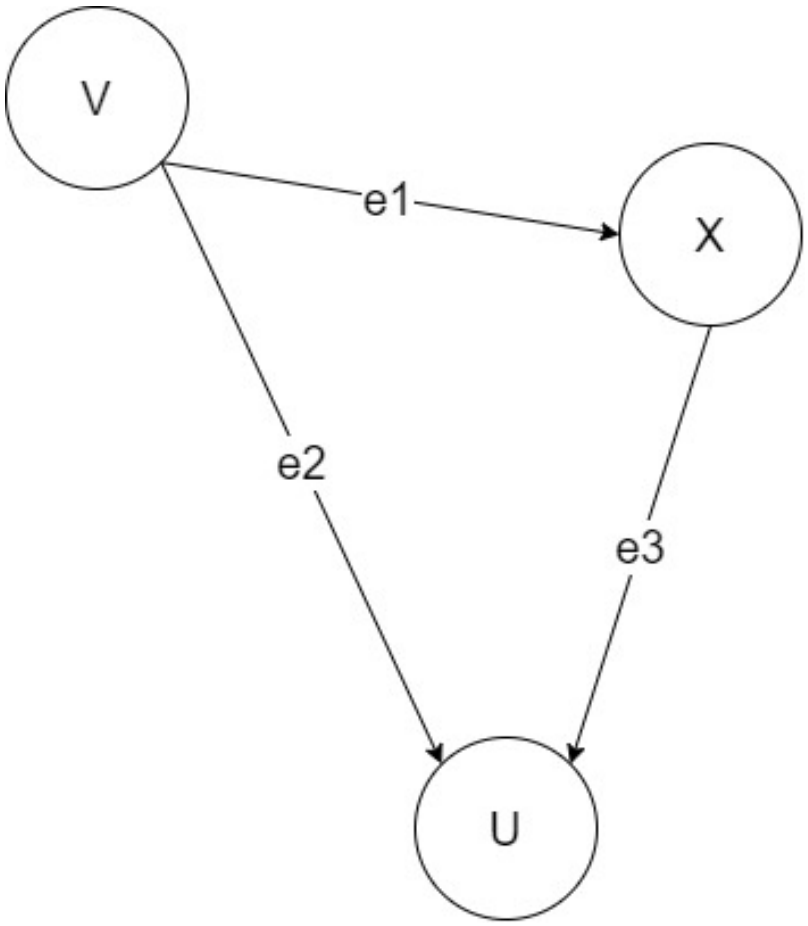


Figura 7.7: Exemplo de conexão direta e alternativa.

Uma vez que temos a conexão direta e alternativa em mãos, avaliamos qual é a mais barata. Entender este ponto do algoritmo fica muito mais simples quando pensamos em escalas de voo. Pense que você quer ir do Rio de Janeiro para Anta Gorda, no Rio Grande do Sul (acredite em mim quando digo que esta cidade existe), para visitar aquela sua tia que você não vê há muito tempo. Contudo, a passagem área direta é muito cara e não cabe no seu orçamento, então você começa a procurar alternativas e percebe que se fizer uma escala via São Paulo (capital) acaba saindo muito barato do que voar diretamente. Logo, a soma destas etapas intermediárias dá o valor total do trajeto Rio de Janeiro - Anta Gorda. Esta é uma situação que acontece frequentemente para quem precisa usar a malha área, e cabe perfeitamente no nosso cenário.

Muitas vezes pode sair mais barato ir de `v` para `u` passando por `x` do que ir diretamente. Portanto, se verdadeira, será considerado como o custo total de `v` para `u` a soma do trajeto completo passando por `x`. Este é o cerne do algoritmo e entendê-lo é crucial para sua implementação em nosso projeto

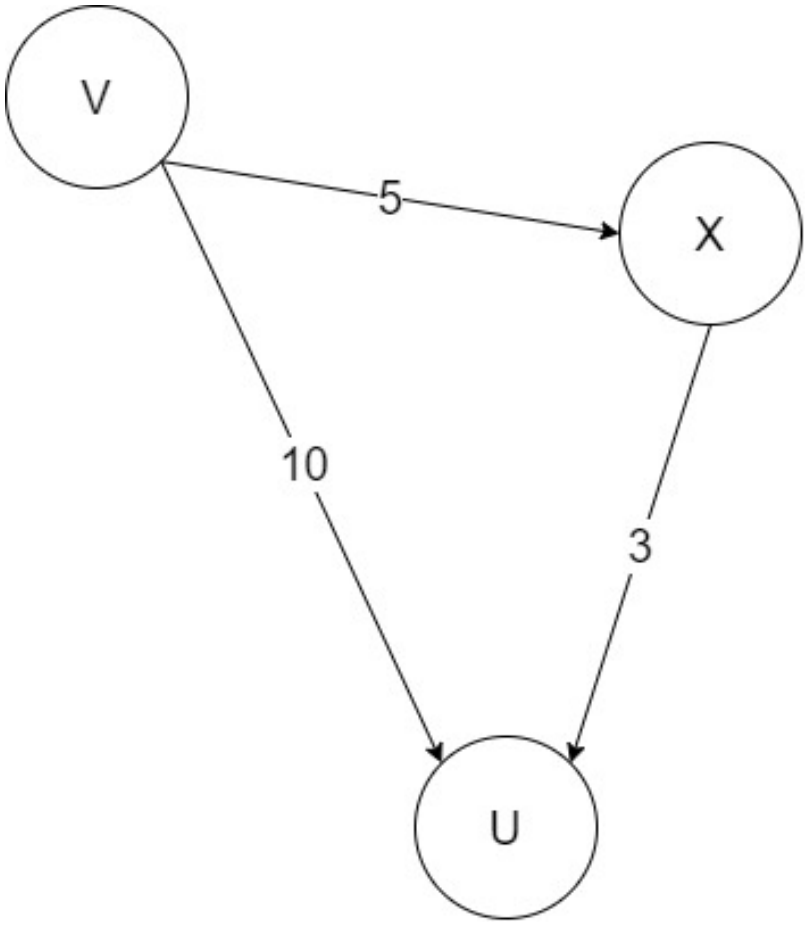


Figura 7.8: Usar uma conexão alternativa sai mais barato neste contexto.

Uma vez que percorremos o grafo em sua completude e fizemos todas as checagens e modificações necessárias, o algoritmo termina.

Avante à parte técnica

Assim como no algoritmo de Dijkstra, onde criamos uma classe exclusiva que guarda toda sua implementação, aqui não será

diferente. Crie a classe `AlgoritmoFloydWarshall` dentro do pacote `util`.

```
src > main > java > grafo > util > AlgoritmoFloydWarshall.java

package main.java.grafo.util;

public class AlgoritmoFloydWarshall { }
```

Esta classe também utilizará uma abordagem *singleton*, como fez a classe `AlgoritmoDijkstra`. Adicione o código a seguir.

```
src > main > java > grafo > util > AlgoritmoFloydWarshall.java

package main.java.grafo.util;

public class AlgoritmoFloydWarshall {
    private static AlgoritmoFloydWarshall algoritmoFloydWarshall;

    private AlgoritmoFloydWarshall(){}

    public static AlgoritmoFloydWarshall getInstance(){
        if(algoritmoFloydWarshall == null){
            return new AlgoritmoFloydWarshall();
        }
        return algoritmoFloydWarshall;
    }
}
```

O que nos resta é adicionar o método `processar(digrafo)` responsável por executar o algoritmo em um dígrafo e retornar uma matriz otimizada. Adicione o código a seguir.

```
src > main > java > grafo > util > AlgoritmoFloydWarshall.java

package main.java.grafo.util;

public class AlgoritmoFloydWarshall {
    /*trechos de códigos já adicionados*/

    public Map<String, Map<String, Info>> processar(Digrafo digrafo){
```

```
        Map<String, Map<String, Info>> matriz = new HashMap<>();
        for(Vertice u : digrafo.getVertices()){
            Map<String, Info> linha = new HashMap();
            matriz.put(u.getRotulo(), linha);
            for(Vertice v : digrafo.getVertices()){
                int peso = digrafo.getPeso(u.getRotulo(), v.getRotulo());
                int valor = peso == 0 ? Integer.MAX_VALUE : peso;
                Info info = new Info();
                info.porQualVertice = v;
                info.distancia =
u.getRotulo().equalsIgnoreCase(v.getRotulo()) ? 0 : valor;
                linha.put(v.getRotulo(), info);
            }
        }

        for(Vertice k : digrafo.getVertices()){
            Map<String, Info> linhaK = matriz.get(k.getRotulo());
            for(Vertice u : digrafo.getVertices()){
                Map<String, Info> linhaU = matriz.get(u.getRotulo());
                Info uk = linhaU.get(k.getRotulo());
                for(Vertice v : digrafo.getVertices()){
                    Info kv = linhaK.get(v.getRotulo());
                    /*
                    Essa verificação é necessária pois operações que
envolvam o valor Integer.MAX_VALUE resultam em valores negativos
                    devido à extrapolação do valor que um dado do tipo
inteiro pode assumir.
                    Veja o link abaixo para maiores detalhes:

https://softwareengineering.stackexchange.com/questions/323292/why-adding-
positive-values-in-java-or-c-sometimes-result-in-a-negative-value
                     */
                    int soma = uk.distancia == Integer.MAX_VALUE ||
kv.distancia == Integer.MAX_VALUE ? Integer.MAX_VALUE : uk.distancia +
kv.distancia;
                    if(soma < linhaU.get(v.getRotulo()).distancia){
                        Info info = new Info();
                        info.porQualVertice = uk.porQualVertice;
                        info.distancia = soma;
                        linhaU.put(v.getRotulo(), info);
                    }
```

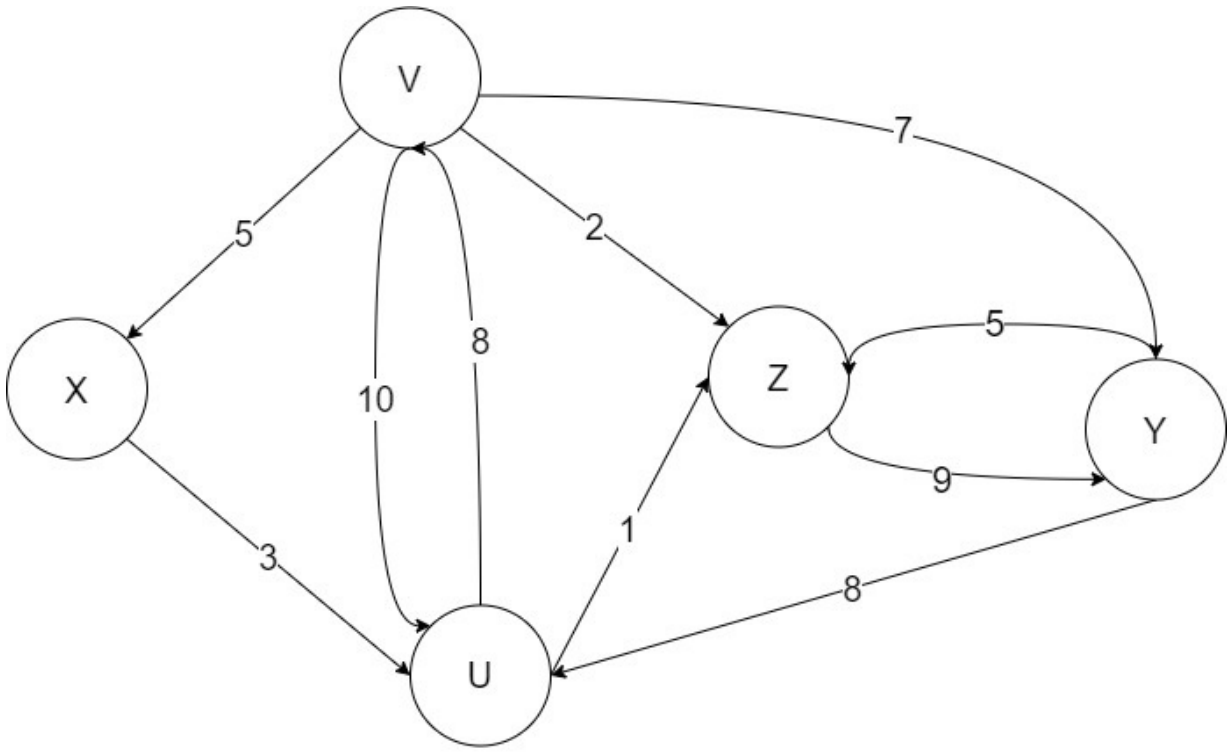
```
                }
            }
        }

        return matriz;
    }
}
```

O método `processar(digrafo)` pode ser dividido em duas etapas: inicialização e processamento. A primeira alimenta a segunda e corresponde ao primeiro bloco de código, que inicializa a matriz retornada. Essa inicialização é feita levando em conta todos os vértices, estando eles conectados por alguma aresta ou não.

Para casos em que existe uma aresta entre dois vértices, o peso da aresta é passado à matriz na célula que corresponde essa conexão, mas quando não há, um valor "infinito" (`Integer.MAX_VALUE`) é usado com o intuito de simbolizar que a distância entre os dois vértices é incalculável, ou seja, inatingível.

Para o grafo da imagem a seguir, a etapa de inicialização produziria a seguinte matriz.



	U	V	X	Z	Y
U	U (0)	V (8)	X (∞)	Z (1)	Y (∞)
V	U (10)	V (0)	X (5)	Z (2)	Y (7)
X	U (3)	V (∞)	X (0)	Z (∞)	Y (∞)
Z	U (∞)	V (∞)	X (∞)	Z (0)	Y (9)
Y	U (8)	V (∞)	X (∞)	Z (5)	Y (0)

Figura 7.9: Matriz produzida pela etapa de inicialização.

Uma vez que a matriz foi construída inicia-se a segunda etapa. É analisado para cada combinação entre dois vértices se existe algum outro vértice que possa agir como intermediário. Se o trajeto incluindo o intermediário for menor que o direto, então a matriz é atualizada. No fim, uma matriz otimizada é retornada.

	U	V	X	Z	Y
U	U (0)	V (8)	V (13)	Z (1)	Z (10)
V	X (8)	V (0)	X (5)	Z (2)	Y (7)
X	U (3)	U (11)	X (0)	U (4)	U (13)
Z	Y (17)	Y (25)	Y (30)	Z (0)	Y (9)
Y	U (8)	U (16)	U (21)	Z (5)	Y (0)

Figura 7.10: Matriz otimizada produzida pela etapa de processamento.

Assim como na classe `AlgoritmoDijkstra` também foi necessário criar uma *inner class* para auxiliar no processamento do algoritmo, aqui não seria diferente. Crie a *inner class* `Info` como mostra o código.

`src > main > java > grafo > util > AlgoritmoFloydWarshall.java`

```
package main.java.grafo.util;

public class AlgoritmoFloydWarshall {
    /*trechos de códigos já adicionados*/

    public Map<String, Map<String, Info>> processar(Digrafo digrafo){
        /*conteúdo do método*/
```

```
    }

    public class Info {
        public int distancia;
        public Vertice porQualVertice;
    }
```

Pronto! Já está pronto para uso. A seguir veja um exemplo.

```
Digrafo digrafo = new Digrafo();

/*crio vértices*/
/*crio arestas*/

Map<String, Map<String, AlgoritmoFloydWarshall.Info>> matriz =
AlgoritmoFloydWarshall.getInstance().processar(digrafo);
for(String v : matriz.keySet()){
    System.out.println("Vértice " + v);
    Map<String, AlgoritmoFloydWarshall.Info> linha = matriz.get(v);
    for(String u : linha.keySet()){
        AlgoritmoFloydWarshall.Info info = linha.get(u);
        System.out.println(u + " com distância " + info.distancia + " por
" + info.porQualVertice.getRotulo());
    }
    System.out.println();
}
```

7.3 O algoritmo de Prim

De volta ao capítulo 5, tivemos nosso primeiro contato com as árvores e aprendemos suas características, assim como a forma de se montar uma. Esta forma utilizava um método já aprendido do capítulo 4.

Entretanto, nosso conhecimento sobre árvores não deve se ater unicamente ao conteúdo visto. Neste momento devemos expandir

nossas mentes em direção a um conceito mais profundo que nos leve a responder mais perguntas que ainda permanecem.

E uma dessas perguntas é: seria possível construir uma árvore T a partir de um grafo ponderado G, sendo que essa árvore deve possuir todos os vértices de G e esses vértices devem permanecer conectados com arestas que possuam o menor peso possível? Será que existe tal coisa?

Se você parar para olhar bem, essa pergunta nos faz relembrar das árvores geradoras que vimos no capítulo 5. O conceito meio que se encaixa aqui, exceto pela parte "possuir o menor peso possível". Neste contexto escolher arestas com pesos mínimos nos levará a menores comprimentos (*length*).

Respondendo à pergunta "será que existe tal árvore", sim, existe. E não somente existe, como é o resultado do algoritmo de Prim, uma árvore geradora mínima.

Uma árvore geradora mínima, ou também conhecida em inglês como *Minimum Spanning Tree*, é uma derivação do conceito da árvore geradora. Afinal de contas, o nome e o conceito não têm semelhanças à toa. A meta principal e as condições necessárias para existência de uma árvore geradora continuam valendo, visto que uma árvore é derivada da outra. O único diferencial entre conceitos é que para árvores geradoras mínimas é necessário encontrar uma forma de manter a árvore conectada, mas por arestas que possuam o menor peso possível, ou seja, o mínimo (Even, Shimon; 1979).

DEFINIÇÃO

Uma árvore T só pode ser considerada a árvore geradora mínima de um grafo G se respeita todas as condições necessárias de uma árvore geradora e é a que possui menor peso total (soma de todos os pesos de arestas) dentre todas as possíveis árvores geradoras de G.

O funcionamento

Sua ideia em linhas gerais é bem simples: fazer crescer uma árvore T a partir de um vértice inicial s, sua raiz, adicionando gradativamente novas folhas que foram obtidas através da execução de um percurso que sempre leva a um vértice por uma aresta com o menor peso. Este processo deve ser executado até todos os vértices do grafo estarem na árvore (Even, Shimon; 1979).

Para que o processo fique claro em nossa mente, vamos aplicá-lo ao grafo a seguir.

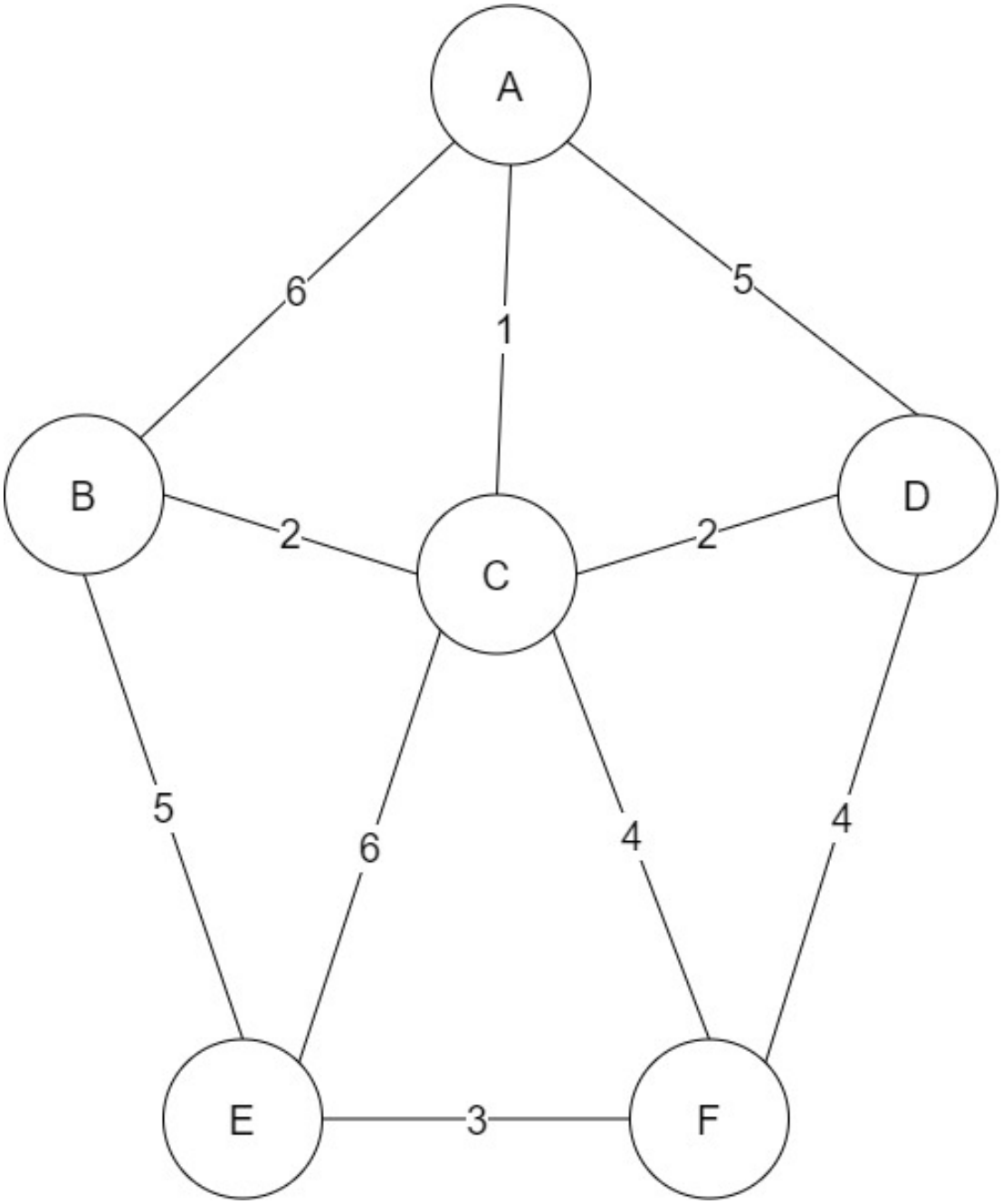


Figura 7.11: Grafo G para aplicação do algoritmo de Prim.

Seja o vértice A a raiz. Ele é adicionado à árvore e suas conexões com vértices adjacentes são analisadas como candidatas a exploração. É eleita a conexão com a menor distância como candidata. Na imagem acima, essa é a conexão AC .

Agora, para cada vértice que ainda não faz parte da árvore, isto é, `B`, `C`, `D`, `E` e `F`, é eleito entre os candidatos existentes (neste momento só um - `AC`) qual deles representa a melhor conexão, ou seja, a conexão que possui a menor distância.

Uma vez eleito, os seguintes passos acontecem:

- `C` é adicionado à árvore;
- `A` é conectado a `C` por uma aresta com peso igual àquele representado pelo candidato eleito;
- `C` é removido do conjunto de vértices ainda não pertencentes à árvore;
- O processo de análise de candidatos recomeça para `A` pois `C` já foi adicionado à árvore e ainda existem vértices adjacentes não analisados;
- O processo de análise de candidatos é feito para `C` em cima de seus adjacentes que ainda não estão na árvore.

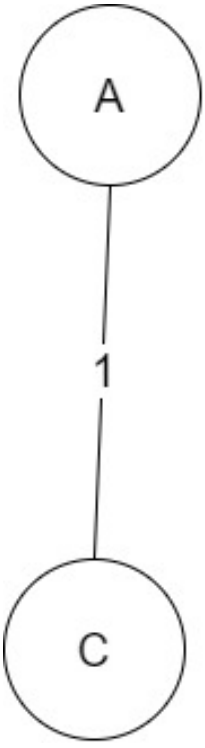


Figura 7.12: Etapa 1 - Árvore geradora mínima gerada por Prim.

Todo o processo é feito novamente, exceto pela adição da raiz à árvore e a primeira análise de candidatos. As conexões AD e CB são marcadas como candidatas sendo CB a melhor porque o peso de sua aresta é dois. B é adicionado à árvore e é conectado a C por uma aresta com peso igual a dois. B é removido dos vértices ainda não presentes na árvore e os candidatos são analisados levando em conta C e B.

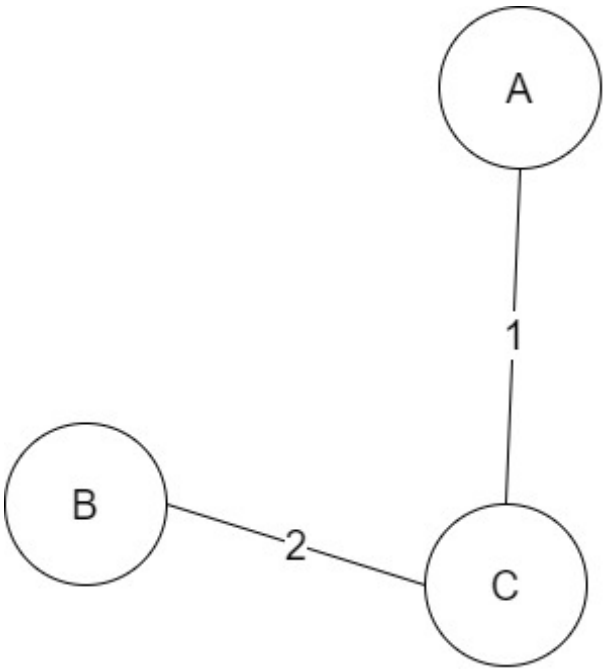


Figura 7.13: Etapa 2 - Árvore geradora mínima gerada por Prim.

Novos candidatos BE e CD são encontrados, AD permanece e CD é eleito o melhor entre eles. D é adicionado à árvore, C e D são conectados por uma aresta com peso igual 2, D é removido dos vértices ainda não presentes na árvore e os candidatos são analisados levando em conta C e D.

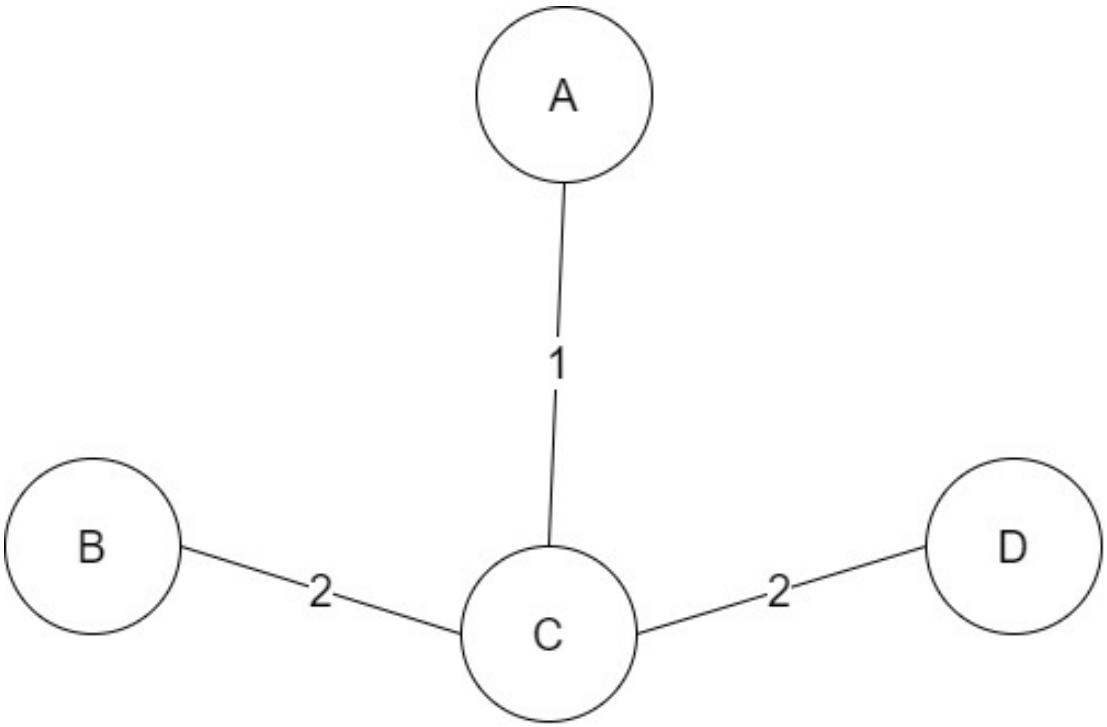


Figura 7.14: Etapa 3 - Árvore geradora mínima gerada por Prim.

Novos candidatos CF e DF são encontrados, AD e BE permanecem e CF é eleito o melhor entre eles. F é adicionado à árvore, C e F são conectados por uma aresta com peso igual 4, F é removido dos vértices ainda não presentes na árvore e os candidatos são analisados levando em conta C e F.

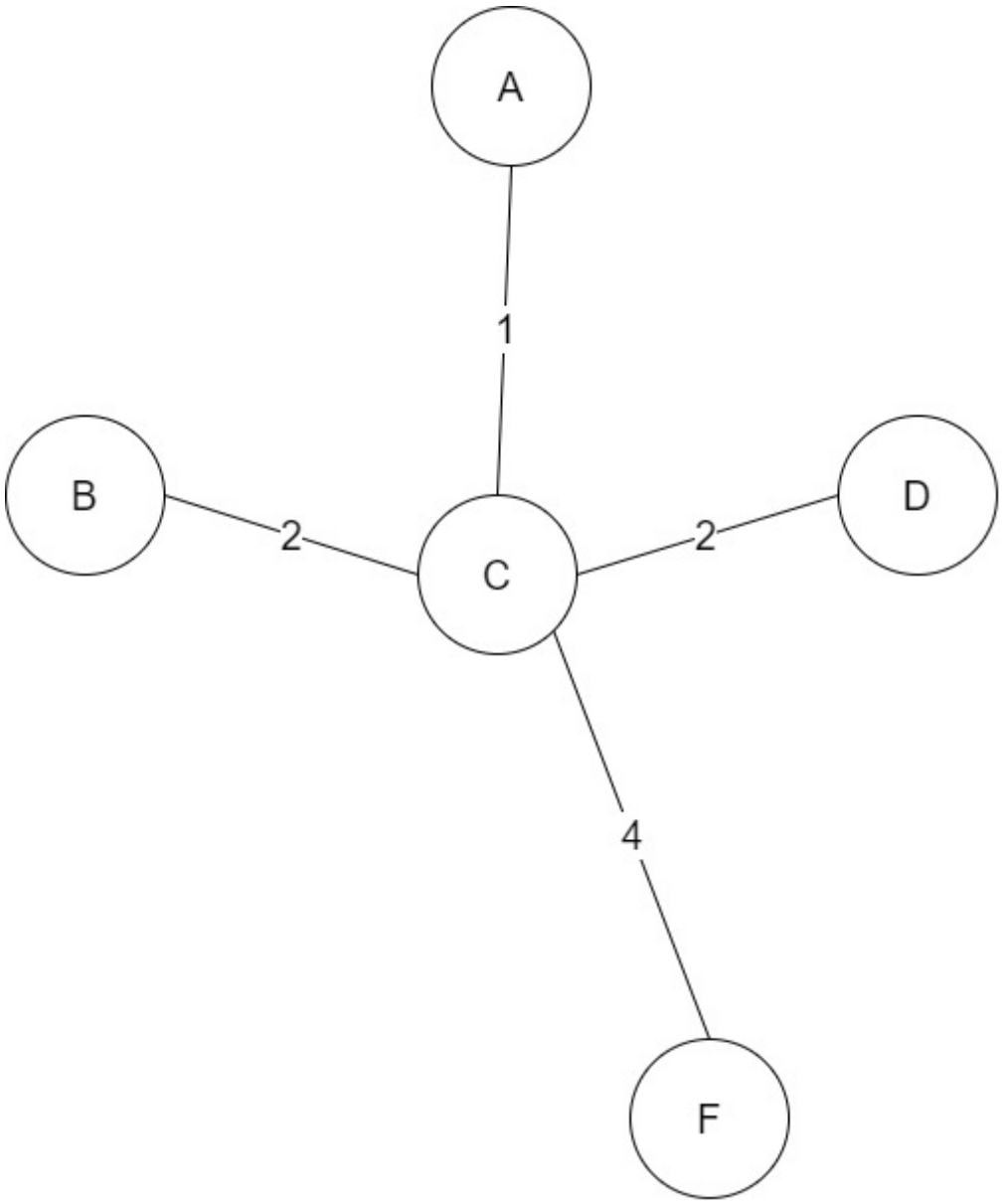


Figura 7.15: Etapa 4 - Árvore geradora mínima gerada por Prim.

Esta é última etapa do algoritmo pois sabemos que só falta o vértice E a ser adicionado. Novos candidatos CE e FE são encontrados, AD, BE e DF permanecem e FE é eleito o melhor entre eles. E é adicionado à árvore, F e E são conectados por uma aresta com peso igual 3, E é removido dos vértices ainda não presentes na árvore e agora todos os vértices do grafo já se encontram na árvore. Os candidatos são analisados levando em conta F e E embora não haja mais necessidade. O algoritmo chega ao fim.

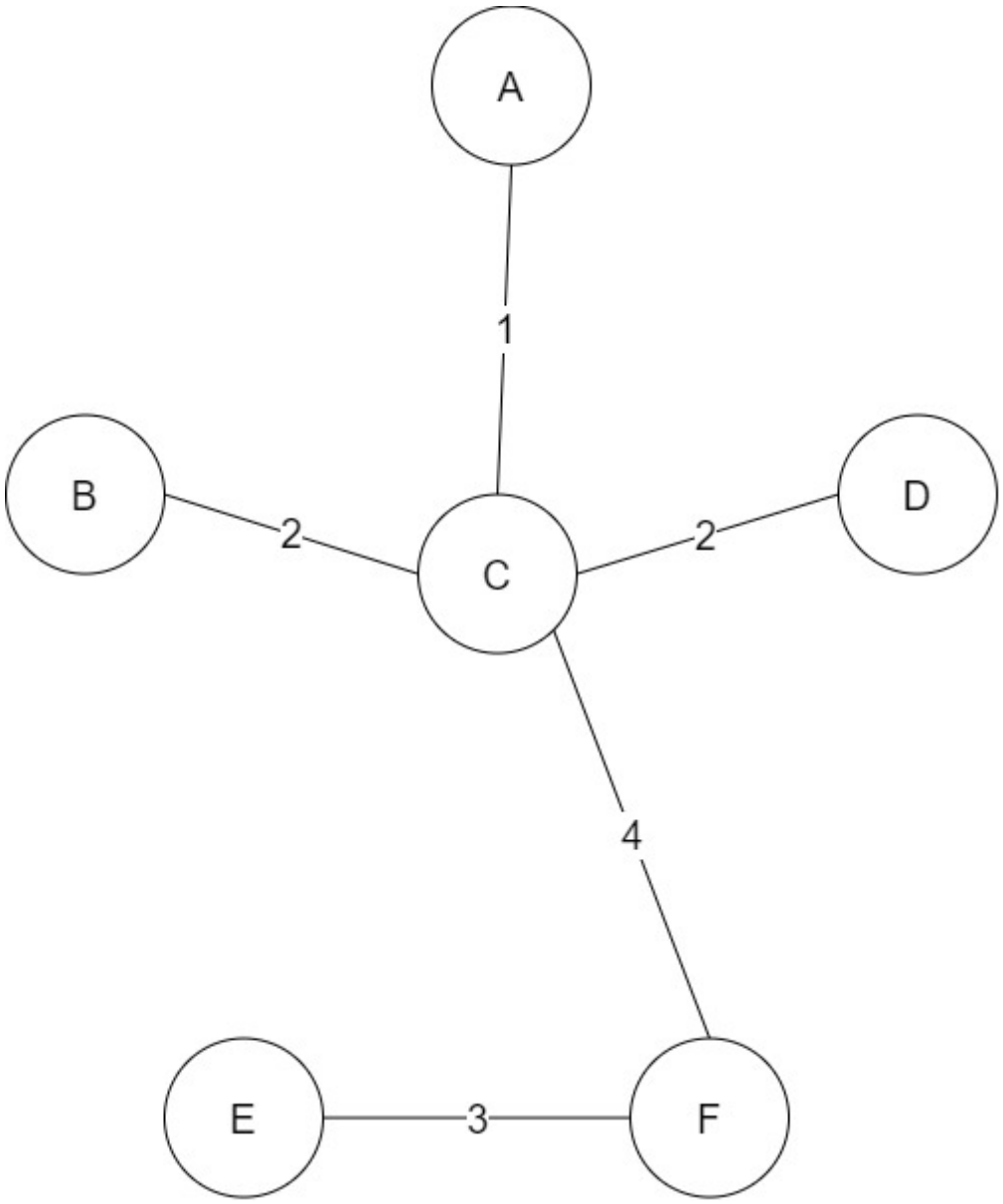


Figura 7.16: A árvore geradora mínima gerada pelo algoritmo de Prim.

CURIOSIDADES

Os algoritmos de Dijkstra e Prim se parecem, mas historicamente Prim veio antes e foi redescoberto por Dijkstra depois (Even, Shimon; 1979).

Aplicar o algoritmo de Prim em um grafo que possua todos seus pesos iguais é uma forma ineficiente de construir uma MST, pois neste cenário qualquer árvore geradora é "mínima" (Lafore, Robert; 2004).

Construindo a implementação

Para transformar todo o funcionamento e conceito visto nas duas últimas subseções é necessário construir uma classe somente para este fim. Ela deverá existir dentro do pacote `util` assim como as outras classes deste capítulo. Crie a classe `AlgoritmoPrim` .

```
src > main > java > grafo > util > AlgoritmoPrim.java

package main.java.grafo.util;

public class AlgoritmoPrim { }
```

Alguns dos elementos dessa classe já são conhecidos e por tal motivo dispensam explicações. Adicione.

```
src > main > java > grafo > util > AlgoritmoPrim.java

package main.java.grafo.util;

public class AlgoritmoPrim {
    private static AlgoritmoPrim algoritmoPrim;

    private AlgoritmoPrim(){}

    public static AlgoritmoPrim getInstance(){
        if(algoritmoPrim == null){
```

```
            return new AlgoritmoPrim();
        }
        return algoritmoPrim;
    }
}
```

Adicione o método `processar(raiz, grafo)`. Seu retorno é um dígrafo representando a árvore geradora mínima construída a partir da raiz e do grafo passados como parâmetros.

`src > main > java > grafo > util > AlgoritmoPrim.java`

```
package main.java.grafo.util;

public class AlgoritmoPrim {
    /*resto da classe*/

    /*novos atributos*/
    private Map<String, String> candidatos;
    private Set<String> aConectar;

    public Digrafo processar(String raiz, Grafo grafo) throws Exception {
        aConectar = new HashSet<>();
        for(Vertice vertice : grafo.getVertices()){
            aConectar.add(vertice.getRotulo());
        }

        Digrafo mst = new Digrafo();
        mst.adicionarVertice(raiz);
        aConectar.remove(raiz);

        candidatos = new HashMap();
        atualizarCandidatos(grafo, raiz);

        while(aConectar.size() > 0){
            String melhorU = null;
            String melhorV = null;
            int menorDistancia = Integer.MAX_VALUE;
            for(String u : candidatos.keySet()){
                String v = candidatos.get(u);
                int peso = grafo.getPeso(u, v);
```

```
                if(peso < menorDistancia){
                    melhorU = u;
                    melhorV = v;
                    menorDistancia = peso;
                }
            }

            if(menorDistancia == Integer.MAX_VALUE)
                break;

            mst.adicionarVertice(melhorV);
            mst.conectarVertices(melhorU, melhorV, menorDistancia);
            aConectar.remove(melhorV);
            atualizarCandidatos(grafo, melhorU);
            atualizarCandidatos(grafo, melhorV);
        }

        return mst;
    }
}
```

É interessante analisar o método `processar` junto da parte explicativa do algoritmo (subseção "O funcionamento") porque desta forma fica muito mais fácil entender. Alguns pontos-chaves merecem destaque:

- Os candidatos são mantidos pelo *map* `candidatos`, cuja chave e valor representam os vértices da conexão;
- O *set* `aConectar` possui os vértices ainda não presentes na árvore;
- A verificação da menor distância (`menorDistancia == Integer.MAX_VALUE`), feita dentro do bloco `while`, serve para casos em que o grafo não é conexo pois pode não haver candidato viável.

O método `atualizarCandidatos(grafo, vertice)` é o responsável pela análise de candidatos comentada anteriormente. Já a eleição deles é feita no primeiro bloco de instruções dentro do *loop* `while` do método `processar`.

src > main > java > grafo > util > AlgoritmoPrim.java

```
private void atualizarCandidatos(Grafo grafo, String vertice){
    int menorDistancia = Integer.MAX_VALUE;
    String maisProximo = null;
    for(Vertice adj : grafo.getAdjacencias(vertice)){
        int peso = grafo.getPeso(vertice, adj.getRotulo());
        if(aConectar.contains(adj.getRotulo()) && peso < menorDistancia){
            menorDistancia = peso;
            maisProximo = adj.getRotulo();
        }
    }
    if(maisProximo != null){
        candidatos.put(vertice, maisProximo);
    } else {
        candidatos.remove(vertice);
    }
}
```

Para usar o algoritmo de Prim em qualquer lugar, adicione o seguinte trecho de código:

```
/*criação do grafo ou dígrafo ponderado*/
/*adição de vértices*/
/*adição de arestas*/
/*outras operações*/

String raiz = "RJ"; //pode ser qualquer raiz
Digrafo mst = AlgoritmoPrim.getInstance().processar(raiz, grafo);
for(Vertice v : mst.getVertices()){
    System.out.println("O vértice " + v.getRotulo() + " é adjacente aos
vértices:");
    for(Vertice adj : mst.getAdjacencias(v.getRotulo())){
        System.out.println(adj.getRotulo() + " com peso " +
mst.getPeso(v.getRotulo(), adj.getRotulo()));
    }
    System.out.println();
    System.out.println();
}
```

7.4 Conclusão

Algoritmos como os de Dikjstra, Floyd e Prim constituem ferramentas importantes para o "canivete suíço" de qualquer um que se aventure pela teoria dos grafos. Juntamente a eles, existem outros trabalhos brilhantes como o de Warshall, Ford ou Kruskal que não foram citados. Acredito que o conteúdo visto aqui foi o suficiente para o nosso contexto, mas é igualmente importante continuar se aprofundando, pois somente assim será possível adicionar mais ferramentas ao seu canivete.

Juntamente com o próximo capítulo que aborda algumas modificações na aplicação, este capítulo constitui o fim de toda uma teoria básica que lhe servirá como um escudeiro em uma jornada que o levará a lugares muito mais distantes e inimagináveis. Enfim, tenha essa singela amostra de conhecimento como uma semente de curiosidade que germina em sua mente.

Resumo

- O algoritmo de Dijkstra trata do problema do menor caminho de um vértice inicial para todos os outros vértices do grafo.
- A ideia do algoritmo de Dijkstra é a manutenção de dois conjuntos T e P, onde T é formado por vértices temporariamente marcados, isto é, menores caminhos que foram calculados de forma não definitiva, ainda podendo mudar se for encontrada uma aresta que resulte em um caminho com menor custo. E o conjunto P representa vértices que foram permanentemente marcados, ou seja, aqueles que já tiveram seu menor caminho calculado de forma definitiva.
- O algoritmo de Floyd-Warshall trata do problema do menor caminho para todos os pares. Para cada combinação entre dois vértices do grafo ele determina o menor caminho.

- A ideia do algoritmo de Floyd-Warshall é a procura de trajetos alternativos que sejam mais baratos que conexões diretas para um determinado par de vértices. Caso um trajeto alternativo mais barato seja encontrado, é guardado em uma matriz otimizada.
- A ideia geral do algoritmo de Prim é bem simples, fazer crescer uma árvore a partir de um vértice inicial (sua raiz) e adicionar novas folhas gradativamente que foram obtidas através da execução de um percurso que sempre leva a um vértice por uma aresta com o menor peso. Este processo deve ser executado até todos os vértices do grafo estarem na árvore.
- Os algoritmos de Dijkstra e Prim se parecem, mas historicamente Prim veio antes e foi redescoberto por Dijkstra depois.

Capítulo 8
Um toque final

Toda obra prima precisa daqueles últimos ajustes antes de ser apresentada ao público, e assim é o nosso projeto, um diamante bruto que ainda precisa de um pouco de lapidação. Deixamos diversos pontos soltos durante nossa jornada quando o assunto era apresentação da aplicação para o usuário. Na verdade, nunca paramos o devido tempo para discutir isso. Então, este capítulo é a representação de um esforço final para o encerramento do que viemos montando juntos. Aqui estudaremos a melhor forma de oferecer tudo o que desenvolvemos para quem realmente importa, o usuário.

8.1 Apertando os parafusos

Até este exato parágrafo nunca paramos para nos preocupar com a apresentação de tudo o que foi desenvolvido nos capítulos anteriores a um possível usuário. Só vimos formas de executar o que foi desenvolvido em cada capítulo de uma maneira bem pontual, continuando o que havíamos feito no anterior. Então, chegamos ao momento de dedicar este esforço e finalizar nossa aplicação.

Crie a classe `Aplicacao` dentro do pacote `aplicacao`, e adicione um método `main` nele. O resultado final deve se parecer assim.

```
src > main > java > aplicacao > Aplicacao.java
```

```
package main.java.aplicacao;

public class Aplicacao {
```

```
    public static void main(String[] args) { }
}
```

Nossa aplicação precisa ter cara de aplicação e uma boa forma de iniciar é apresentando uma mensagem de saudação e quais opções o usuário tem. Portanto, crie o método `menu()` logo abaixo do método `main`.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

public class Aplicacao {
    public static void main(String[] args) {
        /*ainda nenhum código aqui*/
    }

    private void menu() {
        StringBuilder texto = new StringBuilder();
        texto   .append("********** Bem vindo ********************\n")
                .append("* Escolha uma das opções abaixo:        *\n")
                .append("* [G]rafo                               *\n")
                .append("* [D]igrafo                             *\n")
                .append("* [V]értice                             *\n")
                .append("* [A]resta                              *\n")
                .append("* Aresta com [Pe]so                     *\n")
                .append("* Busca por [P]rofundidade              *\n")
                .append("* Busca por [L]argura                   *\n")
                .append("* [AG] - Árvore Geradora                *\n")
                .append("* Árvore Geradora Mínima por [Pr]im     *\n")
                .append("* Algoritmo de [Dij]kstra               *\n")
                .append("* Algoritmo de [Fl]oydWarshall          *\n")
                .append("* Gerar representação Graph[Viz]        *\n")
                .append("* [S]air                                *\n")
                .append("************************************\n");
        System.out.print(texto);
    }
}
```

Note o uso dos colchetes para as opções. Tudo o que está entre colchetes representa uma opção, logo, se o usuário deseja criar um

grafo, basta ele digitar **g** ou **G**, e pronto, um novo grafo é criado. Para algumas opções será necessário digitar mais de uma letra, como **viz** ou **dij**. Isto foi necessário para que não houvesse conflito entre as opções.

Bem, já que apresentamos opções ao usuário, devemos ser capazes de ler a opção escolhida. Portanto, crie o método `ler()`, abaixo de `menu()`.

```
src > main > java > aplicacao > Aplicacao.java

package main.java.aplicacao;

public class Aplicacao {
    public static void main(String[] args) {
        /*ainda nenhum código aqui*/
    }

    private void menu() {
        /*opções*/
    }

    private String ler() {
        return input.next();
    }
}
```

Perceba o uso da variável `input` no método `ler()`. Onde ela foi criada e para que serve? Ela serve para ler o que escrito no console e é do tipo `java.util.Scanner`. Então, para que nossa aplicação não apresente erros de compilação adicione a declaração desta variável fora do método `main`. Sua classe `Aplicacao` deve se parecer assim.

```
src > main > java > aplicacao > Aplicacao.java

package main.java.aplicacao;

import java.util.Scanner;

public class Aplicacao {
```

```
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*ainda nenhum código aqui*/
    }

    private void menu() {
        /*opções*/
    }

    private String ler() {
        /*leitura do que foi digitado no console*/
    }
}
```

Legal! Nossa aplicação não tem erros de compilação, mas o que aconteceria se a executássemos agora? Nada, pois o método `main` ainda está vazio. Então adicione a chamada dos métodos `menu()` e `ler()` para que possamos visualizar o resultado do que fizemos até agora. Seu método `main` deve se parecer assim.

src > main > java > aplicacao > Aplicacao.java

```
public static void main(String[] args) {
    Aplicacao app = new Aplicacao();
    app.menu();
    System.out.print("Digite a opção desejada: ");
    String opcao = app.ler().toUpperCase();
}
```

Execute e veja o que acontece. O menu é exibido, você escolhe uma opção e pronto, o programa chega ao fim. Acho que seria interessante poder escolher qualquer opção mais de uma vez porque assim teríamos a chance de executar o que quisermos. Então, modifique o método `main` para que fique assim.

src > main > java > aplicacao > Aplicacao.java

```
public static void main(String[] args) {
```

```
    Aplicacao app = new Aplicacao();

    app.menu();

    while(true) {
        System.out.print("Digite a opção desejada: ");
        String opcao = app.ler().toUpperCase();
    }
}
```

Dessa forma apresentamos o menu uma única vez e deixamos o usuário livre para escolher quantas opções ele quiser. Mas e se ele escolher uma opção, o que devemos fazer? Devemos executar uma operação de acordo com o que foi escolhido. Modifique o método `main` novamente e adicione após a leitura da opção a seguinte estrutura condicional.

src > main > java > aplicacao > Aplicacao.java

```
public static void main(String[] args) {
    Aplicacao app = new Aplicacao();

    app.menu();

    while(true) {
        System.out.print("Digite a opção desejada: ");
        String opcao = app.ler().toUpperCase();
        switch (opcao) {
            case "G":
                break;
            case "D":
                break;
            case "V":
                break;
            case "A":
                break;
            case "PE":
                break;
            case "P":
                break;
```

```
            case "L":
                break;
            case "AG":
                break;
            case "PR":
                break;
            case "DIJ":
                break;
            case "FL":
                break;
            case "VIZ":
                break;
            case "S":
            default:
        }
    }
}
```

Note as instruções `case` com diversas letras combinadas e que essas combinações correspondem exatamente ao que é mostrado no menu de opções, na mesma ordem. Pois bem, dentro de cada `case` adicionaremos chamadas a funções que executarão exatamente o que foi solicitado pelo usuário.

A primeira opção, e assim primeira instrução `case` , é a opção **G**. Esta opção significa um novo grafo. Caso um grafo já tenha sido criado e esta opção for executada novamente, o grafo antigo é sobrescrito pelo novo. Adicione o método `novoGrafo()` como mostrado a seguir.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.Scanner;

import main.java.grafo.core.Grafo;

public class Aplicacao {
```

```
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        Aplicacao app = new Aplicacao();
        Grafo grafo = null;

        app.menu();

        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                case "G":
                    grafo = app.novoGrafo();
                    break;
                /*continuação*/
            }
        }
    }

    private void menu() {
        /*opções*/
    }

    private String ler() {
        /*leitura do que foi digitado no console*/
    }

    private Grafo novoGrafo() {
        Grafo grafo = new Grafo();
        System.out.println("Novo grafo criado.");
        return grafo;
    }
}
```

O método `novoGrafo()` retorna um novo grafo quando chamado. Como será em um grafo que todas as operações posteriores serão executadas, precisamos guardá-lo e por esse motivo a variável `grafo` foi criada.

Caso **D** tenha sido pressionado, um novo dígrafo deve ser criado. Crie o método `novoDigrafo()` como mostrado a seguir. Seu funcionamento é bem parecido com o do método `novoGrafo()`.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.Scanner;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;

public class Aplicacao {

    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        Aplicacao app = new Aplicacao();
        Grafo grafo = null;

        app.menu();

        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                case "G":
                    /*código aqui*/
                case "D":
                    grafo = app.novoGrafo();
                    break;
            }
        }
    }

    private void menu() { ... }
    private String ler() { ... }
    private Grafo novoGrafo() { ...}
```

```
    private Digrafo novoDigrafo() {
        Digrafo digrafo = new Digrafo();
        System.out.println("Novo dígrafo criado.");
        return digrafo;
    }
}
```

A terceira opção, criação de vértices, é acionada caso seja pressionado **V**. Esta opção somente deve ser executada caso um grafo ou dígrafo já tenha sido criado pois, caso contrário, um erro será causado e aplicação será encerrada. Adicione o método `novoVertice(Grafo grafo)` como mostrado a seguir na classe `Aplicacao`.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.Scanner;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;
import main.java.grafo.core.Vertice;

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                case "G":
                    /*código aqui*/
                case "D":
                    /*código aqui*/
                case "V":
                    app.novoVertice(grafo);
                    break;
```

```
                /*mais cases aqui*/
            }
        }
    }

    private void menu() { ... }
    private String ler() { ... }
    private Grafo novoGrafo() { ...}
    private Digrafo novoDigrafo() { ... }

    private void novoVertice(Grafo grafo) throws Exception {
        System.out.print("Defina o nome do vértice ? ");
        String nome = ler();
        grafo.adicionarVertice(nome);
        System.out.println("Novo vértice " + nome + " criado.");
    }
}
```

As opções **A** e **Pe** estão intimamente relacionadas e serão apresentadas juntas. Respectivamente são: a inclusão de aresta sem peso e a inclusão de aresta com peso. Adicione os métodos `novaAresta(grafo)` e `novaArestaPonderada(grafo)`, e suas chamadas ao método `main` como mostrado adiante.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.Scanner;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;
import main.java.grafo.core.Vertice;

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
```

```
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                case "G":
                case "D":
                case "V":
                case "A":
                    app.novaAresta(grafo);
                    break;
                case "PE":
                    app.novaArestaPonderada(grafo);
                    break;
                /*mais cases aqui*/
            }
        }
    }

    private void menu() { ... }
    private String ler() { ... }
    private Grafo novoGrafo() { ...}
    private Digrafo novoDigrafo() { ... }
    private void novoVertice(Grafo grafo) throws Exception { ... }

    private void novaAresta(Grafo grafo) throws Exception {
        System.out.print("Qual o vértice de origem ? ");
        String vOrigem = ler();
        System.out.print("Qual o vértice de destino ? ");
        String vDestino = ler();
        grafo.conectarVertices(vOrigem, vDestino, null);
        System.out.println("Nova aresta criada.");
    }

    private void novaArestaPonderada(Grafo grafo) throws Exception {
        System.out.print("Qual o vértice de origem ? ");
        String vOrigem = ler();
        System.out.print("Qual o vértice de destino ? ");
        String vDestino = ler();
        System.out.print("Qual o peso da aresta ? ");
        String peso = ler();
        grafo.conectarVertices(vOrigem, vDestino, Integer.valueOf(peso));
        System.out.println("Nova aresta ponderada criada.");
```

```
    }
}
```

Chegamos neste exato momento a um marco divisor em relação às funcionalidades da aplicação, pois da opção **Pe** em diante virão opções que operarão em cima do grafo com seus vértices e arestas. Essas opções compreendem buscas, algoritmos que computam menores caminhos, árvores etc.

Com tudo o que foi criado até o momento na classe `Aplicacao` é possível criar qualquer tipo de grafo ou dígrafo já visto em capítulos anteriores. Recomendo que você passe um tempo fazendo este exercício.

Assim como as opções **A** e **Pe** estão relacionadas, também acontece com as opções **P** e **L** porque ambas representam buscas, respectivamente, profundidade e largura. Adicione os métodos `buscarPorProfundidade(grafo)` e `buscarPorLargura(grafo)` como mostrado a seguir.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.List;
import java.util.Scanner;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;
import main.java.grafo.core.Vertice;
import main.java.grafo.search.BuscaEmLargura;
import main.java.grafo.search.BuscaEmProfundidade;

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
```

```
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                /*mais cases aqui*/
                case "PE":
                    ...
                case "P":
                    app.buscarPorProfundidade(grafo);
                    break;
                case "L":
                    app.buscarPorLargura(grafo);
                    break;
            }
        }
    }

    private void menu() { ... }
    /* mais métodos */
    private void novaArestaPonderada(Grafo grafo) throws Exception { ... }

    private void buscarPorProfundidade(Grafo grafo) {
        System.out.println("Busca por Profundidade");
        System.out.print("Qual o vértice de origem ? ");
        String inicio = ler();
        System.out.print("Qual o vértice de destino ? ");
        String fim = ler();
        List<String> caminhoPercorrido =
BuscaEmProfundidade.getInstance().buscar(grafo, inicio, fim);
        graphVizParaBuscas(grafo, caminhoPercorrido);
    }

    private void buscarPorLargura(Grafo grafo) {
        System.out.println("Busca por Largura");
        System.out.print("Qual o vértice de origem ? ");
        String inicio = ler();
        System.out.print("Qual o vértice de destino ? ");
        String fim = ler();
        List<String> caminhoPercorrido =
BuscaEmLargura.getInstance().buscar(grafo, inicio, fim);
        graphVizParaBuscas(grafo, caminhoPercorrido);
    }
}
```

Não se importe com o método `graphVizParaBuscas(grafo, caminhoPercorrido)` pois ele será explicado mais tarde.

As opções **AG** e **Pr** também têm uma relação íntima, visto que ambas geram árvores a partir do grafo corrente. A primeira extrai uma árvore geradora e a segunda, uma árvore geradora mínima pelo algoritmo de Prim. Adicione os métodos `arvoreGeradora(grafo)` e `arvoreGeradoraMinimaPorPrim(grafo)` , como mostrado a seguir.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

import java.util.List;
import java.util.Scanner;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;
import main.java.grafo.core.Vertice;
import main.java.grafo.search.BuscaEmLargura;
import main.java.grafo.search.BuscaEmProfundidade;

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                /*mais cases aqui*/
                case "L":
                    ...
                case "AG":
                    app.arvoreGeradora(grafo);
                    break;
                case "PR":
                    app.arvoreGeradoraMinimaPorPrim(grafo);
                    break;
```

```
                /*mais cases aqui*/
            }
        }
    }

    private void menu() { ... }
    /* mais métodos */
    private void buscarPorLargura(Grafo grafo) { ... }

    private void arvoreGeradora(Grafo grafo) throws Exception {
        System.out.println("Árvore Geradora");
        Grafo arvore = grafo.arvoreGeradoraPorProfundidade();
        String graphViz = this.graphViz(grafo, false);
        boolean isDigrafo = grafo instanceof Digrafo ? true : false;
        graphVizParaArvores(isDigrafo, arvore, graphViz, false);
    }

    private void arvoreGeradoraMinimaPorPrim(Grafo grafo) throws Exception
{
        System.out.println("Árvore Geradora por Prim");
        System.out.print("Qual a raiz ? ");
        String raiz = ler();
        Digrafo arvore = AlgoritmoPrim.getInstance().processar(raiz,
grafo);
        String graphViz = this.graphViz(arvore, true);
        boolean isDigrafo = grafo instanceof Digrafo ? true : false;
        graphVizParaArvores(isDigrafo, arvore, graphViz, true);
    }
}
```

Assim como para o método `graphVizParaBuscas(grafo, caminhoPercorrido)`, será mais à frente que os métodos `graphVizParaArvores` e `graphViz` serão esclarecidos.

As opções **Dij** e **Fl** representam, respectivamente, a aplicação do algoritmo de Dijkstra e de Floyd-Warshall. Para isso, é necessário adicionar os métodos `algoritmoDijkstra(grafo)` e `algoritmoFloydWarshall(grafo)` e tudo o que os acompanha.

src > main > java > aplicacao > Aplicacao.java

```java
package main.java.aplicacao;

import java.util.List;
import java.util.Map;
import java.util.Scanner;
import java.util.regex.Pattern;

import main.java.grafo.core.Grafo;
import main.java.grafo.core.Digrafo;
import main.java.grafo.core.Vertice;
import main.java.grafo.search.BuscaEmLargura;
import main.java.grafo.search.BuscaEmProfundidade;
import main.java.grafo.util.AlgoritmoDijkstra;
import main.java.grafo.util.AlgoritmoDijkstra.Info;
import main.java.grafo.util.AlgoritmoFloydWarshall;
import main.java.grafo.util.AlgoritmoPrim;

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                /*mais cases aqui*/
                case "PR":
                    ...
                case "DIJ":
                    app.algoritmoDijkstra(grafo);
                    break;
                case "FL":
                    app.algoritmoFloydWarshall(grafo);
                    break;
                /*mais cases aqui*/
            }
        }
    }
```

```
    private void menu() { ... }
    /* mais métodos */
    private void arvoreGeradoraMinimaPorPrim(Grafo grafo) throws Exception
{ ... }

    private void algoritmoDijkstra(Grafo grafo) {
        System.out.println("Algoritmo de Dijkstra");
        System.out.print("Qual o vértice de origem ? ");
        String origem = ler();
        System.out.print("Qual o vértice de destino ? ");
        String destino = ler();
        Map<String, Info> menorCaminho =
AlgoritmoDijkstra.getInstance().processar(origem, destino, grafo);
        String graphViz = this.graphViz(grafo, true);
        for(String key : menorCaminho.keySet()) {
            Info info = menorCaminho.get(key);
            if(info.predecessor != null) {
                String conexaoAsString = info.predecessor.getRotulo() + "
-> " + key;
                int peso = grafo.getPeso(info.predecessor.getRotulo(),
key);
                String regex = conexaoAsString + "\\[.*\\]";
                String replacement = conexaoAsString + "[color=red,label="
+ peso + "]";
                graphViz = graphViz.replaceAll(regex, replacement);
            }
        }
        System.out.println(graphViz);
    }

    private void algoritmoFloydWarshall(Grafo grafo) {
        System.out.println("Algoritmo de Floyd Warshall");
        if(grafo instanceof Digrafo) {
            Digrafo digrafo = (Digrafo) grafo;
            Map<String, Map<String, AlgoritmoFloydWarshall.Info>> matriz =
AlgoritmoFloydWarshall.getInstance().processar(digrafo);
            this.graphVizParaFloydWarshall(matriz);
        } else {
            System.out.println("Operação não permitida para grafos somente
para dígrafos.");
```

```
        }
    }
}
```

A penúltima opção, antes da saída, é a opção responsável por gerar uma representação GraphViz do grafo criado. O GraphViz é um software de visualização de grafos *open source*, possui uma forma de representar grafos e outros objetos através de uma linguagem estrutural e é amplamente conhecido. É possível encontrar na internet diversos sites que possibilitam que seus usuários entrem com suas estruturas GraphViz no formato texto e a partir deles geram representações gráficas. Vale a pena dar uma olhada em https://www.graphviz.org/ pois lá é possível encontrar mais informações e até bibliotecas para linguagens de programação.

Como nosso intuito é nos mantermos o mais simples possível, nossa aplicação vai gerar uma representação GraphViz toda vez que um grafo precise ser impresso. Isso é o que acontece quando os métodos `graphVizParaBuscas`, `graphVizParaArvores`, `graphViz` e `graphVizParaFloydWarshall` são chamados. Todos esses métodos pertencem à classe `Aplicacao`.

Entretanto todos eles possuem uma dependência em comum, o método `graphViz(boolean isPonderado)` da classe `Grafo`, porque ele gera uma representação GraphViz básica de um grafo. Com esta representação básica todos os métodos da classe `Aplicacao` podem realizar suas customizações, inclusive o método `algoritmoDijkstra`. Portanto, adicione na classe `Grafo` o método `graphViz`.

`src > main > java > grafo > core > Grafo.java`

```
package main.java.grafo.core;

/*imports*/
import java.util.regex.Pattern;

public class Grafo {
    /*todos os métodos já desenvolvidos*/
```

```
    public String graphViz(boolean isGrafoPonderado) {
        boolean isDigrafo = this instanceof Digrafo ? true : false;
        StringBuilder graphViz = new StringBuilder();
        graphViz.append("digraph D {\n");

        for(Vertice v : getVertices()) {
            String rotulo = v.getRotulo();
            graphViz.append("\t").append(rotulo).append("
[shape=circle]\n");
            for(Vertice adj : getAdjacencias(rotulo)) {
                String rotuloAdj = adj.getRotulo();
                if(isDigrafo) {
                    graphVizConnection(isGrafoPonderado, isDigrafo,
graphViz, rotulo, rotuloAdj);
                } else {
                    String regex = ".*" + rotuloAdj + " -> " + rotulo +
".*";
                    boolean hasConexaoRedundante =
!Pattern.compile(regex).matcher(graphViz.toString()).find();
                    if (hasConexaoRedundante) {
                        graphVizConnection(isGrafoPonderado, isDigrafo,
graphViz, rotulo, rotuloAdj);
                    }
                }
            }
        }

        graphViz.append("}");
        return graphViz.toString();
    }

    private void graphVizConnection(boolean isGrafoPonderado, boolean
isDigrafo, StringBuilder graphViz, String rotulo,
            String rotuloAdj) {
        graphViz.append("\t\t")
                .append(rotulo).append(" -> ").append(rotuloAdj)
                .append("[");
        if(!isDigrafo) {
            graphViz.append("arrowhead=none");
        }
        if(!isDigrafo && isGrafoPonderado) {
```

```
            graphViz.append(",");
        }
        if(isGrafoPonderado) {
            graphViz.append("label=").append(getPeso(rotulo, rotuloAdj));
        }
        graphViz.append("]\n");
    }
}
```

Agora podemos apresentar os métodos `graphVizParaBuscas`, `graphVizParaArvores`, `graphViz` e `graphVizParaFloydWarshall` da classe `Aplicacao`.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

/*imports*/

public class Aplicacao {
    /*métodos*/

    private String graphViz(Grafo grafo, boolean isPonderado) {
        System.out.println("Para visualizar o grafo acesse o site
http://magjac.com/graphviz-visual-editor/ e cole o conteúdo gerado no
painel da esquerda OU http://www.webgraphviz.com/ e cole o conteúdo gerado
na textarea e clique em \"Generate Graph\".");
        return grafo.graphViz(isPonderado);
    }

    private void graphVizParaBuscas(Grafo grafo, List<String>
caminhoPercorrido) {
        boolean isDigrafo = grafo instanceof Digrafo ? true : false;
        String graphViz = this.graphViz(grafo,false);
        String anterior = null;
        String proximo = null;
        for(String v : caminhoPercorrido) {
            if(anterior == null) {
                anterior = v;
            } else {
                proximo = v;
```

```
                String conexao = anterior + " -> " + proximo;
                String regex = conexao + "\\[.*\\]";

if(!Pattern.compile(regex).matcher(graphViz.toString()).find()) {
                    conexao = proximo + " -> " + anterior;
                    regex = conexao + "\\[.*\\]";
                }
                String replacement = conexao + "[";
                replacement = replacement.concat(!isDigrafo ?
"arrowhead=none,color=red]" : "color=red]");
                graphViz = graphViz.replaceAll(regex, replacement);
                anterior = proximo;
                proximo = null;
            }
        }
        System.out.println(graphViz);
    }

    private void graphVizParaArvores(boolean isDigrafo, Grafo arvore,
String graphViz, boolean isPonderado) {
        for(Vertice v : arvore.getVertices()) {
            for(Vertice adj : arvore.getAdjacencias(v.getRotulo())) {
                String conexaoAsString = v.getRotulo() + " -> " +
adj.getRotulo();
                String regex = conexaoAsString + "\\[.*\\]";
                String replacement = conexaoAsString + "[";
                if(isPonderado) {
                    int peso = arvore.getPeso(v.getRotulo(),
adj.getRotulo());
                    replacement =
replacement.concat("label="+peso).concat(",");
                }
                replacement = replacement.concat(!isDigrafo ?
"arrowhead=none,color=red]" : "color=red]");
                graphViz = graphViz.replaceAll(regex , replacement);
            }
        }
        System.out.println(graphViz);
    }

    private void graphVizParaFloydWarshall(Map<String, Map<String,
```

```
AlgoritmoFloydWarshall.Info>> matriz) {
        System.out.println("Para visualizar o resultado do algoritmo
acesse o site http://magjac.com/graphviz-visual-editor/ e "
                + "cole o conteúdo gerado no painel da esquerda.");
        StringBuilder graphViz = new StringBuilder();
        graphViz.append("digraph D {\n")
                .append("\taHtmlTable [\n")
                    .append("\t\tshape=plaintext\n")
                    .append("\t\tcolor=black\n")
                    .append("\t\tlabel=<\n")
                        .append("\t\t\t<table style=' border: 1px solid
black; text-align: center;'  cellspacing=' 0' >\n");

        graphViz.append("\t\t\t\t<tr>\n\t\t\t\t\t<td></td>\n");
        for(String v : matriz.keySet()) {
            graphViz.append("\t\t\t\t\t<td style=' font-weight:
bold;' >").append(v).append("</td>\n");
        }
        graphViz.append("\t\t\t\t</tr>\n");

        for(String v : matriz.keySet()) {
            graphViz.append("\t\t\t\t<tr>\n")
                    .append("\t\t\t\t\t<td style=' font-weight:
bold;' >").append(v).append("</td>\n");
            Map<String, AlgoritmoFloydWarshall.Info> celulas =
matriz.get(v);
            for(String u : celulas.keySet()) {
                AlgoritmoFloydWarshall.Info info = celulas.get(u);

graphViz.append("\t\t\t\t\t<td>").append(info.porQualVertice.getRotulo()).
append(" (");
                if(info.distancia == Integer.MAX_VALUE) {
                    graphViz.append("8");
                } else {
                    graphViz.append(info.distancia);
                }
                graphViz.append(")").append("</td>\n");
            }
            graphViz.append("\t\t\t\t</tr>\n");
        }
```

```
        graphViz.append("\t\t\t</table>").append("\n\t>];").append("\n}");

        System.out.println(graphViz.toString());
    }
}
```

Todos os métodos da classe `Aplicacao` que geram representações GraphViz são responsáveis por instruir o usuário da aplicação sobre como ele pode visualizar o resultado gerado. Para isso, utilizaremos dois sites: http://magjac.com/graphviz-visual-editor/ e http://www.webgraphviz.com/. Ambos apresentam uma área em que possível adicionar a estrutura GraphViz em formato de texto gerada pela aplicação, e, após inserida, realizar a transformação para a representação gráfica.

Como dito em parágrafos anteriores, a penúltima opção antes da saída imprime o grafo corrente. Então, caso o usuário escolha **Viz** o método `imprimir(grafo)` é chamado.

src > main > java > aplicacao > Aplicacao.java

```
package main.java.aplicacao;

/*imports*/

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                /*mais cases aqui*/
                case "FL":
                    app.algoritmoFloydWarshall(grafo);
                    break;
                case "VIZ":
                    app.imprimir(grafo);
```

```
                    break;
                /*case "S" aqui*/
            }
        }
    }

    /*métodos*/
    private void graphVizParaFloydWarshall(Map<String, Map<String,
AlgoritmoFloydWarshall.Info>> matriz) { ... }

    private void imprimir(Grafo grafo) {
        System.out.println("Imprimindo grafo...");
        System.out.print("O grafo é ponderado (s/n) ? ");
        String resposta = ler();
        String graphViz = this.graphViz(grafo,
"s".equalsIgnoreCase(resposta));
        System.out.println(graphViz);
    }
}
```

E assim chegamos à última opção, a saída. Esta opção é mais simples de todas, pois ela encerra a aplicação informando um código de sucesso. Adicione o método `sair()` e sua chamada ao método `main`.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

/*imports*/

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
        /*código aqui*/
        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            switch (opcao) {
                /*mais cases aqui*/
```

```
                case "VIZ":
                    ...
                case "S":
                    System.out.println("Encerrando...");
                    app.input.close();
                    app.sair();
                default:
                    System.out.println("Opção não reconhecida.");
                    app.input.close();
                    app.sair();
            }
        }
    }

    /*métodos*/
    private void imprimir(Grafo grafo) { ... }

    private void sair() {
        System.out.println("Programa finalizado.");
        System.exit(0);
    }
}
```

Existe aqui uma pegadinha, porque se quisermos executar nossa aplicação agora não será possível. Existem erros de compilação, mas onde? Adicionamos vários métodos à classe `Aplicacao` que são suscetíveis a erros (*exceptions*), onde os tratamos? Até agora, em lugar nenhum. Então modifique o método `main` para que ele fique da forma mostrada a seguir.

`src > main > java > aplicacao > Aplicacao.java`

```
package main.java.aplicacao;

/*imports*/

public class Aplicacao {
    private Scanner input = new Scanner(System.in);

    public static void main(String[] args) {
```

```
        Aplicacao app = new Aplicacao();
        Grafo grafo = null;

        app.menu();

        while(true) {
            System.out.print("Digite a opção desejada: ");
            String opcao = app.ler().toUpperCase();
            try{
                switch (opcao) {
                    /*cases e default aqui*/
                }
            } catch (Exception e) {
                System.out.println("Erro: " + e.getMessage());
                System.out.println("Abortando ...");
                System.out.println("Programa finalizado.");
                System.exit(1);
            }
        }
    }

    /*métodos*/
}
```

E este passo encerra as modificações feitas na classe `Aplicacao` para suportar a criação/uso dos grafos desenvolvidos em capítulos anteriores. Execute a aplicação como um todo agora e veja como ela se comporta.

Assim terminamos a primeira versão da aplicação. Ainda existe muito trabalho a ser feito, e é hora de você trilhar seus próprios passos pois esta obra chegou ao fim. Entretanto, existem alguns pontos que podem ser melhorados e outros que podem ser ainda mais explorados. Aventure-se em outras dimensões da Teoria dos Grafos e incorpore novas funcionalidades à aplicação. O horizonte à sua frente é infinito. Boa sorte! :)

8.2 Conclusão

Finalizamos os últimos pontos que faltavam para que pudéssemos entregar um produto viável ao usuário final e vimos a importância de criar uma interface amigável para um uso apropriado da ferramenta. Sem ela, todo o trabalho desenvolvido nos capítulos anteriores seria em vão pois nunca teríamos a chance de ver a aplicação funcionando. Nosso objetivo sempre foi entregar uma ferramenta na qual o usuário pudesse decidir o que fazer, uma vez apresentadas as opções, e assim extrair resultados. Com o término deste capítulo atingimos isso.

Detalhes ainda faltam? Claro que sim, mas isso faz parte de um novo ciclo no desenvolvimento de um software. Finalizamos a construção de uma longa lista de funcionalidades e agora entramos em um ciclo de aperfeiçoamento delas. O que precisa ser feito cabe a você e ao seu usuário decidir. Você agora possui um olhar mais técnico e crítico, visto que conhece os alicerces da aplicação. O desenvolvimento de um software é incremental, sempre vamos adicionando de pouco em pouco, tijolo por tijolo.

Capítulo 9
O epílogo de uma aventura

Caro leitor, cara leitora, parabéns! Você chegou ao fim dessa jornada, que podemos considerar uma "quase odisseia". Se estivéssemos em um jogo esta seria a parte onde os créditos sobem na tela e você espera pelas cenas pós-créditos.

Além disso, este capítulo representa uma conversa direta entre nós dois, uma conclusão final de toda esta obra. Aqui pretendo explicar certas motivações e escolhas que me guiaram durante o processo de construção deste trabalho.

Antes de mais nada, devo lhe agradecer com todo o meu coração por ter adquirido este livro, seja ele no formato impresso ou e-book ou os dois, e por ter depositado sua confiança em mim fazendo com que pudesse ser o seu guia por um mundo completamente novo aos seus olhos.

Uma pergunta que sempre me fiz durante todo este processo foi: qual será minha contribuição com este livro para quem lê e para mundo? Não me interprete mal mas esta pergunta é justificável uma vez que tive contato com trabalhos de Even Shimon, Harary, Bondy & Murty, entre outros. Então, uma vez que você nada entre tubarões e tem consciência de que ainda é e talvez sempre será um peixinho, você se faz esse tipo de pergunta. Depois de muita reflexão, cheguei a uma resposta.

Claramente, esta obra não estará à altura dos trabalhos que citei no último parágrafo pois eles possuem um incrível arcabouço teórico/acadêmico necessário para propor novos algoritmos e estruturas em grafos. Contudo, o que criei toma força, e consequentemente forma, quando volto minha atenção para as pessoas que estão começando a explorar a área. Evidentemente, não estou afirmando que tais obras citadas não tiveram a mesma

preocupação, mas conforme fui me aprofundando nesse estudo, notei que a linguagem matemática era intensa e isso pode se tornar uma barreira para quem é iniciante. Então é aqui que resolvi atacar.

Li e reli o material que reuni e através de um processo repetitivo e muitas vezes cansativo fui debulhando todo este conhecimento em explicações detalhadas com um linguajar mais descontraído, para que pudesse agregar algum conhecimento tanto para quem já conhece alguma coisa da área, assim como para quem não conhecia nada. No fim das contas, este trabalho representa uma luta árdua contra o preconceito com a matemática no geral. Foi assim que tive meu primeiro contato com a teoria dos grafos, através de um professor na minha graduação que conseguia transferir e traduzir um conhecimento complexo em uma forma límpida e leve para nossas mentes. Portanto, era esse o meu gol.

Um outro ponto em que também acho que fiz a diferença foi a aplicação desenvolvida em Java. Até hoje somente encontrei um livro que abordava grafos e conseguia entregar uma aplicação prática através de uma linguagem de programação, ou seja, unir o teórico ao prático. Essa obra foi a de Robert Lafore e através dela fui inspirado.

A escolha de usar Java como linguagem de programação para o livro é uma questão pessoal. Já trabalho com esta linguagem há muito tempo e gosto dela então já me sinto familiarizado, mas, como disse no primeiro capítulo, nada impede que você implemente o mesmo projeto em outra linguagem.

Acredito que uma abordagem voltada para o uso de JavaScript seria até melhor porque JavaScript é mais didático que Java. Java é uma linguagem mais verbosa e a verbosidade nesse momento é algo que nos atrapalha, então é bom evitar o uso de recursos muito avançados, eleva a complexidade e tira o foco do verdadeiro desafio, os grafos.

Outro ponto que é interessante trazer a tona aqui é o direcionamento que o livro teve perante algumas decisões. Durante a leitura você deve ter se perguntando por que não expliquei certas coisas, por que não montei uma aplicação mais rebuscada usando técnicas da web, ou por que deixei certos detalhes técnicos passarem.

Entenda uma coisa, quando se monta qualquer tipo de material deve se criar certos limites. Esses limites representam a linha até onde o seu material pode ir. Caso essa linha não seja imposta, o material tende a crescer mais e mais com detalhes que o distanciam do verdadeiro alvo.

Finalmente, gostaria de deixar uma mensagem final como parte da contribuição total deste livro. Nunca desista de um sonho, independente qual seja. Não se assuste com determinados temas, por mais que pareçam impossíveis. Antes, dedique-se e tente, arrisque, deixando medos e preconceitos de lado. Procure suporte para dar seus primeiros passos. Não se cobre tanto, tudo tem seu tempo na vida. A lebre não ganhou da tartaruga mesmo sendo mais rápida. Em vez de se preocupar em assimilar o conteúdo com velocidade, preocupe-se em prestar mais atenção em detalhes, em ser mais minucioso. Não se prenda a certos valores, tenha sua mente e ouvidos sempre abertos ao que existe de novo. Todos nós estamos em uma jornada, cada um em sua parte dela. O simples nem sempre é fácil, lembre-se disto.

O fim deste capítulo representa para mim o encerramento de uma aventura e de uma parte da minha vida, para você o início da uma nova jornada. Existem ainda muitos assuntos inexplorados e muitos algoritmos a serem entendidos. A verdadeira aventura está para começar.

Capítulo 10
Referência bibliográfica

Boaventura Netto, Paulo Oswaldo. *Teoria e Modelos de Grafos*. São Paulo: Editora Edgard Blücher, 1979.

Bondy, J. A.; Murty, U. S. R. *Graph Theory with Applications*. University of Waterloo, 5ed, 1976.

Bondy, J. A.; Murty, U. S. R.. *Graph Theory*. Springer, 2008.

Brandão, Souza Junito. *Mitologia Grega*, volume 1. Petrópolis: Editora Vozes, 1986.

Callioli, Carlos; Domingues, Hygino; Costa, Roberto. *Álgebra Linear e Aplicações*, 6ª ed. São Paulo: Editora Atual, 1978.

Cardoso, Antônio. *Uma perspectiva parental sobre a influência das crianças na compra de vestuário*. Faculdade de Ciências Humanas e Sociais - Universidade Fernando Pessoa, Portugal, 2005. ISSN 1646-0502.

Evans, Eric. *Domain-Driven Design*: Atacando as Complexidades no Coração do Software. 3ª ed. Rio de Janeiro: Alta Books, 2017.

Even, Shimon. *Graph Algorithms*. 2ªed. Cambridge University, 1979.

Harary, Frank. *Graph Theory*. Addison-Wesley Publishing Company, 1970.

Lafore, Robert. *Estruturas de Dados e Algoritmos em Java*. Rio de Janeiro: Ciência Moderna, 2004.

Martin, Robert C. *Clean Code*: A Handbook of Agile Software Craftmanship. New Jersey: Prentice Hall, 2009.

Menezes Jr., Rômulo; Machado, Liliane; Medeiros, Álvaro. Aplicação de Algoritmos de Grafos para Gerar e Percorrer Jogos de Labirintos

Aleatórios. *Anais do Congresso de Matemática Aplicada e Computacional*. Universidade Federal da Paraíba - UFPB. CMAC Nordeste, 2012. ISSN 2317-3297

Ochner, Anderson; Pezzini, Anderson. Comparação entre buscas para resolução do jogo resta um. *Revista Científica do Alto Vale do Itajaí - REAVI*. Universidade do Estado de Santa Catarina, 2015; Disponível em: http://www.revistas.udesc.br/index.php/reavi/article/download/5770/4203