UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
DEPARTAMENTO DE MATEMÁTICA
Mestrado Profissional em Matemática em Rede Nacional

Murilo Ramos da Cunha Ribeiro

Grafos, Algoritmos e Programação

RECIFE
2018

UNIVERSIDADE FEDERAL RURAL DE PERNAMBUCO
DEPARTAMENTO DE MATEMÁTICA
Mestrado Profissional em Matemática em Rede Nacional

Murilo Ramos da Cunha Ribeiro

Grafos, Algoritmos e Programação

Dissertação de mestrado apresentada ao Departamento de Matemática da Universidade Federal Rural de Pernambuco como requisito parcial para obtenção do título de Mestre em Matemática.

Orientador: Prof. Dr. Ricardo Nunes Machado Júnior

RECIFE
2018

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema Integrado de Bibliotecas da UFRPE
Biblioteca Central, Recife-PE, Brasil

R484g Ribeiro, Murilo Ramos da Cunha
Grafos, algoritmos e programação / Murilo Ramos da Cunha Ribeiro. – 2018.
152 f.: il.

Orientador: Ricardo Nunes Machado Júnior.
Dissertação (Mestrado) – Universidade Federal Rural de Pernambuco, Programa de Pós-Graduação Mestrado Profissional em Matemática em Rede Nacional, Recife, BR-PE, 2018.
Inclui referências.

1. Matemática – Estudo e ensino 2. Programação (Matemática) 3. Algorítmos 4. Teoria dos grafos 5. Ensino médio I. Machado Júnior, Ricardo Nunes, orient. II. Título

CDD 510

Grafos, Algoritmos e Programação

Murilo Ramos da Cunha Ribeiro

Dissertação de mestrado APROVADA, em 19/10/2018, apresentada ao Departamento de Matemática da Universidade Federal Rural de Pernambuco como requisito parcial para obtenção do título de Mestre em Matemática.

Orientador:

Prof. Dr. Ricardo Nunes Machado Júnior

Banca examinadora:

Prof. Dr. Marcelo Pedro dos Santos
UFRPE

Prof. Dr. Eudes Naziazeno Galvão
UFPE

RECIFE

2018

Dedico esse trabalho à minha família em toda sua composição, à minha esposa Rafaella Karolyne, aos meus pais Severina Ramos e Claudio Ribeiro, às minhas tias e tios, primas e primos, sobrinhos e sobrinhas, e aos filhos e filhas, netos e netas que virão.
Dedico também aos amigos e irmãos que a vida me apresentou, Victor Hugo e Ana Catarine; aos que guerrearam junto comigo no PROFMAT, os amigos da minha turma, que fortaleciam uns aos outros, dividindo o pouco conhecimento que tinham, para multiplicar as vitórias que conquistamos; aos mestres e doutores da UFRPE que nos lançaram os seus mais nobres conhecimentos.

Agradecimentos

À Deus, escritor, projetista e engenheiro do Universo. À minha família. Ao doutor orientador Ricardo Nunes, pelas palavras essenciais e a orientação imensurável à conclusão desse trabalho. Às energias do Universo, que conspiram para o bem, mesmo quando tudo parece perdido.

O RIO E O OCEANO

Diz-se que mesmo antes de um rio cair no oceano ele treme de medo.

Olha para trás, para toda a jornada, os cumes, as montanhas, o longo caminho sinuoso através das florestas, através dos povoados, e vê à sua frente um oceano tão vasto que entrar nele nada mais é do que desaparecer para sempre.

Mas não há outra maneira.
O rio não pode voltar. Ninguém pode voltar.
Voltar é impossível na existência.
Você pode apenas ir em frente.
O rio precisa se arriscar e entrar no oceano.
E somente quando ele entra no oceano é que o medo desaparece.
Porque apenas então o rio saberá que não se trata de desaparecer no oceano, mas tornar-se oceano.
Por um lado é desaparecimento e por outro lado é renascimento.

Assim somos nós.
Só podemos ir em frente e arriscar.
Coragem!!!
Avance firme e torne-se Oceano!!!

Osho

Declaração

Eu, **Murilo Ramos da Cunha Ribeiro**, declaro para devidos fins e efeitos, que a dissertação sob título **Grafos, Algoritmos e Programação**, entregue como Trabalho de Conclusão de curso para obtenção do título de mestre, com exceção das citações diretas e indiretas claramente indicadas e referenciadas, é um trabalho original. Eu estou consciente que a utilização de material de terceiros incluindo uso de paráfrase sem a devida indicação das fontes será considerado plágio, e estará sujeito à processo administrativos da Universidade Federal Rural de Pernambuco e sanções legais. Declaro ainda que respeitei todos os requisitos dos direitos de autor e isento a Pós-graduação PROFMAT/UFRPE, bem como o orientador **Dr. Ricardo Nunes Machado Júnior**, de qualquer ônus ou responsabilidade sobre a sua autoria.

Recife, 19 de Outubro de 2018.

Murilo Ramos da Cunha Ribeiro

Resumo

Nesse trabalho de dissertação, propomos que seja inserida a Teoria dos Grafos no Ensino Médio, fazendo uso do *software* SageMath juntamente com o uso de programação como ferramenta na busca de soluções matemáticas em problemas que pareçam cansativos tomar a solução de forma manual, além de propor uma sequência didática para o ensino de Teoria dos Grafos no Ensino Médio fazendo uso de algoritmos e programação. Essa sequência didática é voltada para alunos de 2º ano do Ensino Médio, porém as ferramentas que aqui discutiremos podem abranger quaisquer outras séries com os devidos cuidados.

Palavras-chave: Algoritmos, Grafos, Programação.

Abstract

In this dissertation, we propose to insert the Theory of Graphs in High School, making use of SageMath software together with the use programming as a tool in the search for mathematical solutions in problems that seem tiresome to take the solution manually, in addition to proposing a didactic sequence for the teaching of Graph Theory in High School making use of algorithms and programming. This didactic sequence is aimed for students in the second year of High School, but the tools that we will discuss here can cover any other series with the proper care.

Key-word: Algorithms, Graphs, Programming.

Lista de ilustrações

Lista de tabelas

Lista de Códigos

Sumário

Introdução

O ensino de Combinatória nas nossas escolas tem se limitado de forma usual aos problemas de contagem. Apesar dos Parâmetros Curriculares Nacionais (PCN) não contemplarem o ensino de Teoria dos Grafos nas escolas, as Orientações Curriculares Nacionais (BRASIL, 2006, p. 94) defende a exploração de outros conteúdos matemáticos.

O documento argumenta que:

> Outros tipos de problemas poderiam ser trabalhados na escola - são aqueles relativos a conjuntos finitos e com enunciados de simples entendimento relativo, mas não necessariamente fáceis de resolver. Um exemplo clássico é o problema das pontes de Könisberg. (BRASIL, 2006, p. 94)

Partindo dessa necessidade de explorar novos horizontes, as Orientações Curriculares afirma ainda que:

> Problemas dessa natureza podem ser utilizados para desenvolver uma série de habilidades importantes: modelar o problema, via estrutura de grafo - no exemplo, um diagrama em que cada ilha é representada por um ponto e cada ponte é um segmento conectando dois pontos; explorar o problema, identificando situações em que há ou não solução; convergir para a descoberta da condição geral de existência de uma tal solução (ainda no exemplo, o caso em que cada ilha tem um número par de pontes). Muitos outros exemplos de problemas combinatórios podem ser tratados de modo semelhante, tais como determinar a rota mais curta em uma rede de transportes ou determinar um eficiente trajeto para coleta de lixo em uma cidade. (BRASIL, 2006, p. 94)

Em se tratanto de tecnologia, sabemos do desafio constante de ensinar matemática em um mundo globalizado, onde a tecnologia, que está acessível na palma da mão, vem ganhando proporções assustadoras com a variedade de aplicativos que prometem fazer de tudo como, por exemplo, solucionar uma equação apenas apontando a câmera do *smartphone*.

Propõe-se aqui o uso despretencioso da prática de programação no ambiente escolar como alternativa para despertar no aluno o interesse pela matemática e pelo exercício do pensamento lógico, pautados na nova redação da Base Nacional Curricular Comum (BNCC), que reforça o uso tecnologias digitais de comunicação e informação de forma crítica, significativa, reflexiva e ética nas diversas práticas do cotidiano (incluindo as escolares) ao se comunicar, acessar e disseminar informações, produzir conhecimentos e resolver problemas".

As Orientações Curriculares Nacionais (BRASIL, 2006, pp. 87-90) defendem o uso da tecnologia no ensino da matemática, estimulando, por exemplo, o uso de planilhas eletrônicas no ensino de funções, o uso de *software* de geometria dinâmica no ensino de geometria enriquecendo e aprimorando as imagens mentais associadas às propriedades geométricas.

As Orientações Curriculares Nacionais afirmam ainda:

> É com a utilização de programas que oferecem recursos para a exploração de conceitos e idéias matemáticas que está se fazendo um interessante uso de tecnologia para o ensino da Matemática. Nessa situação, o professor deve estar preparado para interessantes surpresas: é a variedade de soluções que podem ser dadas para um mesmo problema, indicando que as formas de pensar dos alunos podem ser bem distintas; a detecção da capacidade criativa de seus alunos, ao ser o professor surpreendido com soluções que nem imaginava, quando pensou no problema proposto; o entusiástico engajamento dos alunos nos trabalhos, produzindo discussões e trocas de idéias que revelam uma intensa atividade intelectual. (BRASIL, 2006, pp. 89-90)

Esse trabalho é voltado a professores do Ensino Médio e alunos da graduação do curso de Licenciatura em Matemática que desejam incluir a Teoria dos Grafos como componente curricular, além do uso da tecnologia em sala de aula.

Dividimos esta dissertação em cinco Capítulos.

No Capítulo inicial discutimos, num primeiro momento, a evolução da tecnologia no mundo e as necessidades de construção do pensamento crítico e investigativo para o aprendizado, além de discutir a importância da programação para o estímulo do nosso cérebro.

Em seguida fazemos uma breve apresentação do *software* SageMath que será o ambiente de aprendizado da prática de grafos e programação no decorrer desse trabalho, mostrando algumas ferramentas básicas desse *software*. Finalizando esse Capítulo, abordamos os conceitos de lógica de programação que será nossa base para a aplicação de programação em sala de aula, bem como introduzimos os comandos básicos da linguagem de programação Python, por ser poderosa e facilmente acessível o seu aprendizado.

No Capítulo 2, abordamos as noções da Teoria dos Grafos, definindo o que é um grafo, bem como apresentando outros conceitos básicos que serão fundamentais para os Capítulos seguintes.

No Capítulo 3, discutimos conceitos de caminhos, grafos conexos, grafos bipartidos, ciclos, grafos eulerianos e hamiltonianos, além da aplicação do problema do menor caminho, muito importante para diversas áreas do conhecimento, sempre conectando à prática dos códigos e programação.

No Capítulo 4, trabalhamos os conceitos de árvores e florestas, com mais ênfase

para o de árvores, além da teoria de árvores geradoras, focando na aplicação do problema de conexão de peso mínimo que visa reduzir custos tornando um processo mais eficiente.

Em todo o tratamento da Teoria dos Grafos, há o trabalho dedicado a programação e apresentação dos comandos e funções já existentes no SageMath que envolve essa área da matemática.

O Capítulo final é composto por uma proposta pedagógica que apresenta uma sequência didática para abordar a Teoria dos Grafos no Ensino Médio, se apoiando no uso da programação, com a presença de exercícios que instiguem o pensamento investigativo no aluno e finalizando com as soluções das atividades propostas.

Esperamos que a experimentação desse trabalho seja coerente com a proposta de discussão pretendida e que o ensino de Teoria dos Grafos, junto ao uso de programação em sala de aula, possa estimular os alunos ao aprendizado.

1 A LÓGICA DE PROGRAMAÇÃO NA CONSTRUÇÃO DO PENSAMENTO MATEMÁTICO

1.1 UM OLHAR SOBRE A EVOLUÇÃO

O surgimento dos computadores e o avanço tecnológico-científico trouxe à tona uma nova Era, a Era Digital. O fluxo de informações que antes caminhavam a passos lentos, hoje possui velocidade assustadora e ganha o mundo em segundos. As tecnologias que antes demoravam décadas para ganhar novas evoluções, hoje ganham novos rumos em curtos prazos, bastando apenas um pouco menos de um trimestre para aquele *smartphone* ganhar uma nova funcionalidade ou melhoria de *hardware* e ser lançado como um novo aparelho, tornando praticamente obsoleto o *smartphone* anterior.

Segundo o livro *História da Computação: O caminho do pensamento e da tecnologia* (FONSECA FILHO, 2007, pp.85-107), a história do computador começa pelo ábaco, um instrumento mecânico de origem chinesa considerado o primeiro computador, que nada mais era do que um tipo simples de calculadora que realizava operações algébricas.

Com o passar do tempo, o grande John Napier, matemático escocês do século XVII, inventou um instrumento analógico capaz de realizar cálculos logarítmicos conhecido como "régua de cálculo".

Mais tarde, veio a surgir a primeira máquina mecânica programável, idealizada pelo matemático francês Joseph-Marie Jacquard, uma espécie de máquina de tear capaz de controlar a confecção dos tecidos através de cartões perfurados.

Já em meados do século XIX, criou-se uma máquina analítica que, de forma grosseira, pode ser comparada com o computador atual com memória e programas, inventada pelo matemático inglês Charles Babbage, considerado o "Pai da Informática".

O computador, palavra que surge do verbo *computar*, que significa *calcular*, ganhou cada vez mais variedade de cálculos matemáticos, permitindo assim que tarefas que pareciam ser repetitivas, difíceis ou que demandassem mais tempo pudessem ser feitas de uma forma mais rápida e eficiente.

As primeiras construções desses computadores, máquinas inicialmente dotadas de válvulas, depois sendo substituídas por transístores e, finalmente, ganhando a tecnologia dos circuitos integrados e microprocessadores, precisava de uma interface homem -

máquina capaz de programar e reprogramar componentes de forma mais simples.

Como todas essas inovações tecnológicas objetivavam a facilitação das atividades e processos comuns realizados pelo homem, era preciso alcançar uma linguagem acessível a todos. Surgiram, então, as primeiras linguagens de programação, conjunto de regras padronizadas e com sentido para instruir uma máquina a realizar uma atividade.

Definição 1 (ALGORITMO)**.** *Algoritmo é um conjunto de regras e operações bem definidas e ordenadas, destinadas à solução de um problema ou de uma classe de problemas, em um número finito de etapas.*

Ao fazer uso de uma linguagem de programação para instruir uma máquina a produzir um resultado esperado, estamos construindo uma sequência lógica de instruções e possibilidades, ou seja, um algoritmo.

Desde muito tempo, o homem resolve os problemas através de algoritmos, um sistema concebido por matemáticos gregos e árabes no século III a.C. e que hoje está presente em várias atividades cotidianas: desde uma pesquisa na internet até a análise de tomografias, radiografias e ressonâncias magnéticas.

Um exemplo de algoritmo é o algoritmo para obter a divisão de dois números naturais com resto, também conhecida como **divisão euclidiana**, ao qual enuncia: sejam a e b dois números naturais, com $a > 0$, existem dois únicos números naturais q e r tais que $b = a \cdot q + r$, com $0 \leq r < a$.

Para obtermos q e r, dados a e b naturais ($a > 0$), usamos a sequência:

- Passo 1: Faça $q = 0$. Temos $r = b$.
- Passo 2: Se $r = b - a \cdot q < a$, pare. Encontramos q e r.
- Passo 3: Se $r = b - a \cdot q \geq a$, faça $q = q + 1$. Volte ao passo 2.

Ainda no princípio do aprendizado de matemática, somos submetidos aos algoritmos básicos das quatro operações, ou seja, aprendemos uma série de passos para alcançar o resultado esperado por essas operações.

Ao programar em determinada linguagem de programação estamos dizendo à máquina o que deve ser feito baseado em uma sequência de códigos, ou seja, em um algoritmo. À medida que programamos, buscamos aperfeiçoar os algoritmos, tornando ainda mais eficientes nossos códigos e assim, alcançar os resultados esperados com muito mais agilidade e perfeição.

Programar nos permite pensar em novas possibilidades, desenvolvendo uma gama de ideias para cada situação. Não se trata apenas de criar um simples algoritmo, mas fomentar um leque de possibilidades em função de uma decisão tomada.

Já que os algoritmos fazem parte do nosso cotidiano de forma tão comum, caberia a nós indagarmos: POR QUE NÃO APRENDEMOS A PROGRAMAR DESDE CEDO?

1.2 A PROGRAMAÇÃO E A CONSTRUÇÃO DO PENSAMENTO LÓGICO

Aprender a programar nos permite desenvolver diversas habilidades cognitivas em qualquer fase da nossa vida: em crianças, excita mais cedo as áreas do lobo frontal, parte do cérebro envolvido no raciocínio lógico, responsável pelo planejamento de ações e pelo pensamento abstrato, desenvolvendo o pensamento computacional e passos lógicos para a resolução de problemas (KAFAI; BURKE, 2013, p.63); nos idosos, a programação estimula o raciocínio e como toda atividade de raciocínio, estimula o exercício físico do cérebro, cumprindo a fundamental função de ajudar a blindar nosso cérebro contra doenças cognitivas, frequente com o avançar da idade (ARAÚJO et al, 2012, p.3).

Com a nova redação da Base Nacional Comum Curricular (BNCC), que foi amplamente discutida com os diversos setores da sociedade brasileira e homologada em 2017, a inserção da tecnologia é tomada como ponto crucial em um mundo cada vez mais movido pela ciência da computação.

Segundo a nova redação da BNCC, dentre as dez competências fundamentais definidas no documento (p.5), uma delas reforça o papel da tecnologia no contexto educacional, a 5ª competência, que afirma ser essencial "utilizar tecnologias digitais de comunicação e informação de forma crítica, significativa, reflexiva e ética nas diversas práticas do cotidiano (incluindo as escolares) ao se comunicar, acessar e disseminar informações, produzir conhecimentos e resolver problemas".

Segundo o relatório DE OLHO NAS METAS 2015-16 do movimento Todos pela Educação [5], em nosso país, apenas 7,3% dos jovens têm aprendizado de matemática adequado à sua série ao final do Ensino Médio. Muitas vezes, isso acontece porque o aluno não enxerga no cotidiano a utilidade daquilo que está sendo repassado em sala de aula. Portanto, a inclusão da tecnologia nos ambientes de aprendizagem daria mais sentido ao que estaria sendo ensinado, constituindo ferramentas necessárias para superar as dificuldades vivenciadas por professores e alunos, visando uma inovação das formas de ensino e aprendizagem, na qual a inclusão de aulas de programação no âmbito escolar tornaria o ensino de outras disciplinas muito mais interessante e cativante (CODE.ORG, 2013).

Entre outros benefícios do ensino de códigos (programação) no ambiente escolar, podemos citar:

- **Lógica Matemática**: Por meio dos códigos e algoritmos, o ato de programar auxilia no estímulo do contato entre a matemática teórica e o seu uso na prática através de atividades exploratórias e contextualização para resolução de problemas. (PONTE, 1991, p.62).

- **Raciocínio Diversificado**: Quando se constrói um algoritmo ou um código, raramente existe "o melhor". Por isso, ao programar, buscamos sempre a eficiência, desenvolvemos a capacidade de decidir, entre várias opções, quais têm custos e benefícios mais vantajosos para determinados objetivos. (RAPKIEWICZ, 2006, p.4).

- **Desenvolvimento do Pensamento Crítico**: Ao criar códigos, trabalhamos a habilidade de se fazer suposições, ao mesmo tempo em que é desenvolvido o raciocínio preciso e específico. Essas vantagens surgem da necessidade das máquinas de executar algoritmos muito específicos, exigindo precisão de quem os executa. (GERALDES, 2014, p.10).

- **Resolução de Desafios**: A programação traz facilidade de lidar com problemas e inventar soluções. Torna seus desenvolvedores aptos na arte de inventar, de fazer descobertas aplicáveis a qualquer tipo de resolução de desafios. (RAPKIEWICZ, 2006, p.4).

Entre tantos benefícios para o desenvolvimento cognitivo, de crianças à idosos, é preciso que haja uma motivação por trás do intuito de aprender a programar. Então como convencer, por exemplo, uma criança de que o aprendizado de uma linguagem de programação é importante para a vida? Como motivá-las a programar?

Esses são alguns questionamentos bem parecidos com o que muitos educadores se deparam quando estão preparando suas aulas, já que não há mais a ideia de que o conhecimento deve ser dado de forma pura, isolado, sem contexto, mas sim, deve-se interagir com outros tópicos e trazê-los para a realidade do alunos, para que este se sinta como protagonista nesse processo de aprendizagem.

Segundo KAFAI e BURKE (2013, p.63), ao aprender sobre algoritmos e programação, as crianças podem agora aprender a programar aplicações específicas, como jogos ou histórias interativas, engajando o potencial deles para criar algo real e tangível à realidade em que vivem, podendo compartilhar suas criações com outras pessoas, convertendo o aprendizado de programação, uma disciplina abstrata, em criações no mundo digital.

No contexto da programação, há uma variedade imensa de linguagens de programação. Não há, de fato, a "melhor" linguagem para programar, pois diante da gama imensa que encontramos, algumas linguagens ganham mais eficiência a depender do objetivo escolhido.

Atualmente, existe uma série de *sites* na internet que buscam contribuir para o aprendizado de programação, com as mais diversas linguagens, com alguns desses sendo totalmente gratuitos, como ***Code Academy*** (< www.codecademy.com/pt >), ***Code.org*** (< www.code.org >), ***Code Avengers*** (< www.codeavengers.com >), dentre outros.

Apesar de gratuitos e com cursos de alta qualidade, convém citar que alguns desses *sites* ensinam apenas a sintaxe das linguagens de programação, isto é, a forma como se escreve os comandos e funções de cada linguagem de programação. É preciso ir ainda mais além: pensar no problema de forma a ser possível quebrá-lo sistematicamente em etapas lógicas e individuais, que permitam atingir o objetivo final desse problema.

1.3 A PODEROSA FERRAMENTA MATEMÁTICA *SAGEMATH*

Um *software* matemático que vem ganhando os usuários é o **SageMath**, nosso objeto de estudo desse trabalho acadêmico. O SageMath é um *software* matemático livre e de código aberto (*open source*), desenvolvido por uma comunidade de programadores e matemáticos, que busca ser uma alternativa para os principais sistemas proprietários de *softwares* matemáticos como o Magma, Maple, Mathematica e Matlab.

Uma das vantagens evidentes de se utilizar *software* livre é a garantia de que qualquer outro pesquisador ou estudante que receba páginas de SageMath será capaz de exibí-las e manipulá-las sem a necessidade de aquisição de *software* proprietário. Além disto qualquer pessoa interessada poderá investigar, e alterar se necessário, o código fonte, situação bem diversa do que ocorre com o *software* proprietário onde o usuário desconhece os algoritmos usados pelo programa.

Antes denominado de Sage e também de SAGE, seu nome teve origem na abreviação de ***S**ystem for **A**lgebra and **G**eometry **E**xperimentation* (ou **S**istema **A**lgébrico e **G**eométrico de **E**xperimentações), ganhou aspectos mais fortes dentre *softwares* matemáticos por integrar vários *softwares* de matemática em uma interface comum, projetando-se para o mercado sob o nome SageMath.

Apesar de ser escrito principalmente em duas linguagens de programação chamadas *Python* e *Cython* [junção da facilidade de programar do *Python* com uma linguagem de programação chamada *C*, que permite rapidez de execução do programa], para o uso mais elaborado do SageMath é exigido que os usuários saibam apenas *Python*. Tanto estudantes quanto profissionais auxiliam na construção e desenvolvimento do SageMath, sendo esse desenvolvimento mantido por trabalhos voluntários e doações.

O SageMath engloba e se utiliza de um grande número de pacotes pré-existentes como *Maxima*, *GAP*, *Pari/GP*, *softwares* de renderização de imagens e muitos outros, integrando-os em uma interface única que busca ser amigável e de fácil assimilação. Isso

significa que esse *software* possui ferramentas que abrangem muitas áreas, incluindo álgebra, combinatória, análise numérica, teoria dos números, recursos gráficos e cálculo. Todos os principais pacotes são instalados juntamente com o SageMath e muitos outros pacotes existem para extensões em áreas específicas.

O SageMath pode ser utilizado por meio de linhas de comandos interativas, fazendo uso de uma interface acionada de dentro de um *browser* [navegadores de *internet* como *Google Chrome*, *Mozilla Firefox* e *Microsoft Edge*], onde os passos são armazenados em páginas separadas por usuário.

Para usar o SageMath remotamente, ou seja, num ambiente em nuvem, o usuário deverá acessar o site <*www.cocalc.com*> e realizar o cadastro para ter acesso ao ambiente do SageMath. Não mostraremos como usar o SageMath instalando localmente, ou seja, instalando normalmente em um computador, mas os procedimentos nos ambientes de trabalho do SageMath permanecem os mesmos. Além disso, mostraremos como usar o SageMath, usando comandos, e apresentaremos algumas de suas ferramentas básicas.

1.4 O AMBIENTE *SAGEMATH* E SUAS FUNCIONALIDADES BÁSICAS

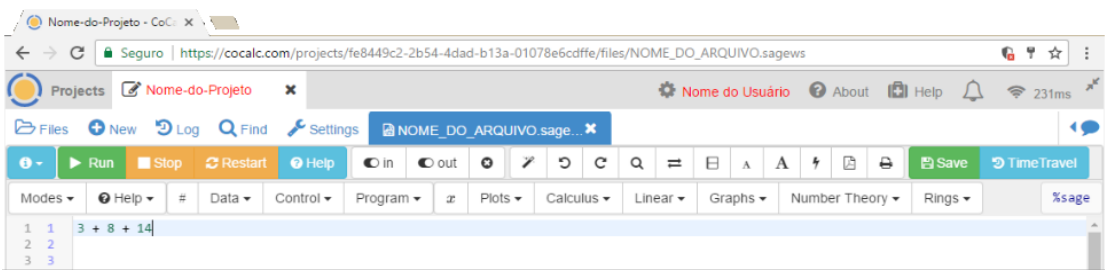
Agora veremos como é simples escrever os códigos em SageMath. Esse *software* traz inúmeras possibilidades de cálculos numéricos, plotagem de gráficos, dentre outras ferramentas.

Quando surgir alguma dúvida de como usar as ferramentas do SageMath, o usuário poderá acessar a documentação do SageMath no site oficial (<*http://www.sagemath.org*>) para saber como usar cada função de forma correta.

Mostraremos aqui alguns exemplos básicos de uso do SageMath:

- Uso como calculadora: é possível executar cálculos comuns às calculadoras. Por exemplo, para somar de valores é só escrever essa soma normalmente e depois clicar no botão "run", conforme a figura abaixo, ou ainda, pressionar SHIFT + ENTER, para executar os cálculos indicados.

Figura 1 – A soma de números no SageMath.

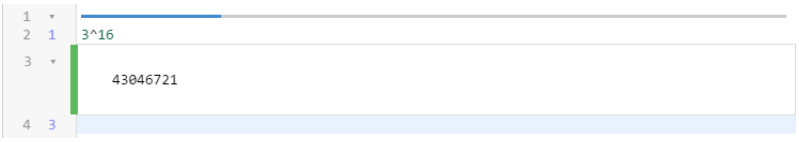


Ao executar o cálculo indicado, obtemos o resultado esperado da soma:

Figura 2 – Resultado da soma de números no SageMath.

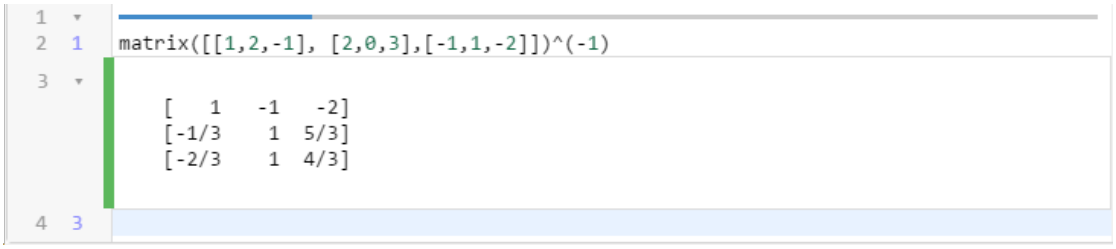
- Para calcular potências, basta usar o acento circunflexo (^) seguido do expoente.

Figura 3 – Resultado da potência 3^{16} no SageMath.



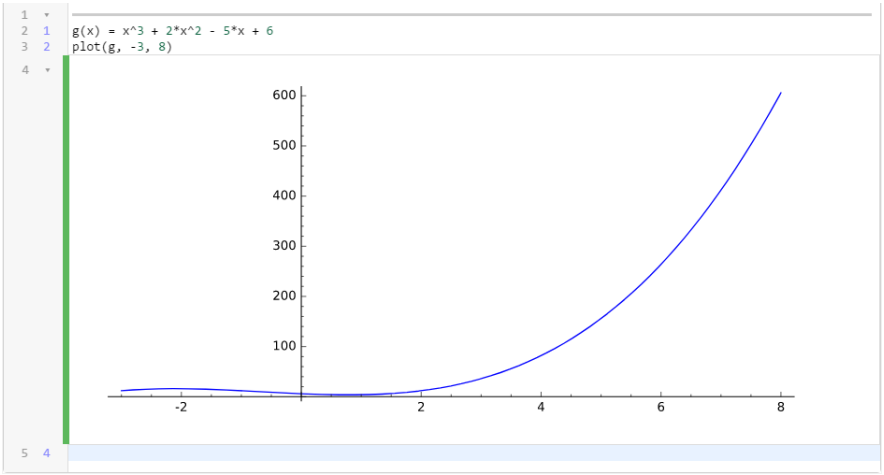
- Para calcular a inversa de uma matriz 3 x 3, basta escrever os elementos das linhas da matriz nas triplas entre colchetes, como mostra o exemplo a seguir: **matrix([[1,2,-1], [2,0,3], [-1,1,-2]])^(-1)**

Figura 4 – Resultado da inversa de uma matriz de ordem 3 no SageMath.



- Para plotar o gráfico de uma função, basta escrever a função como é apresentado no exemplo a seguir e digitar: **plot(nome da função, limite esquerdo do gráfico, limite direito do gráfico)**.

Figura 5 – Gráfico gerado a partir da função $g(x) = x^3 + 2x^2 - 5x + 6$ no SageMath.



Observe que nesse caso, plot(g, -3, 8) significa plotar, desenhar o gráfico da função g, delimitada de $x = -3$ a $x = 8$.

O objetivo desse trabalho é articular o conhecimento de teoria dos grafos com a programação no SageMath, objetivando facilitar a resolução de alguns problemas.

Para isso, trataremos dos conceitos de grafos, problematizando sempre que possível, para que possamos aplicar no SageMath aliados com programação, através de problemas propostos nas atividades do capítulo final desse trabalho, permitindo que os conceitos ganhem um sentido mais palpável.

Na próxima seção, introduziremos os conhecimentos básicos ligados à lógica de programação e à programação em *Python*. O objetivo não é oferecer conhecimentos profundos acerca de lógica de programação, mas apenas mostrar como realizar operações básicas de entrada e saída de dados, estruturas condicionais e estruturas repetitivas.

1.5 NOÇÕES DE LÓGICA DE PROGRAMAÇÃO E COMANDOS BÁSICOS DE PYTHON

1.5.1 **Variáveis, funcões e listas**

Uma **variável** é um espaço na memória do computador onde se armazenam dados que pode ser acessado ou alterado sempre que necessário. Podemos imaginar a memória como um armário cheio de gavetas e quando criamos uma variável estamos reservando uma dessas gavetas para armazenar os dados da variável criada.

Imagine que desejamos criar quatro variáveis: uma chamada "nome" para armazenar o nome de uma pessoa, outra chamada "idade" para guardar a idade dela, outra chamada "altura" para armazenar a altura dessa pessoa em metros e uma outra denominada "notas" para armazenar as notas dela.

Em Python, escrevemos:

Código 1.1 – Declarando variáveis em Python

```
1 nome = "Ariano Suassuna"
2 idade = 87
3 altura = 1.83
4 notas = [5,8,9.5,2]
```

Em Python, o ponto (.) em 1.83 é usado para separar casas decimais.

Esses são exemplos de quatro tipos de dados que a linguagem Python é capaz de processar: texto (*string*), inteiro (*int*), número de ponto flutuante (*float*) e uma lista (*list*), respectivamente.

Em Python, nomes de variáveis podem conter números, porém é obrigatório que você comece com letra ou com o símbolo _.

Tabela 1 – Alguns tipos de variáveis em Python

Tipo	Ao que se refere	Exemplos
string	**a variáveis que armazenam texto**	**fruta = "goiaba"** **cor = "azul"**
int	**a variáveis que armazenam números inteiros**	**num_alunos = 56** **casa = 1**
float	**a variáveis que armazenam números decimais (com "ponto")**	**raiz_aprox_2 = 1.4142** **casa = 1.**
list	**a uma lista com dados**	**conceitos = ['A','B','C','D']** **valores = [15, 78, 41, 12, 18]**

Para exibir valores de variáveis, usamos o comando **print**. Esse comando pode ser usado de duas formas:

- Interpolando as variáveis entre textos como em:

Código 1.2 – Exibindo uma mensagem interpolando variável e texto em Python

```
1 nome = "Joseane Gomes"
2 idade = 15
3 altura = 1.46
4 print "A estudante",nome,"tem",idade,"anos e mede",altura,"metros
      de altura."
```

cujo resultado é dado por:

```
A estudante Joseane Gomes tem 15 anos e mede 1.46000000000000 metros
de altura.
```

- Escrevendo em formato de texto, inserindo o comando %s quando tiver que ser substituído por um texto, %d quando a variável for substituída por um número inteiro, %f quando for substituída por um número real, sendo possível indicar a quantidade de casas decimais como em %.2f onde o número real é apresentado com 2 casas decimais como mostra o exemplo a seguir:

Código 1.3 – Exibindo uma mensagem em formato de texto em Python

```
1 nome = "Joseane Gomes"
2 idade = 15
3 altura = 1.46
4 print "A estudante %s tem %d anos e mede %.2f metros de altura." %(
      nome, idade, altura)
```

cujo resultado é dado por:

```
A estudante Joseane Gomes tem 15 anos e mede 1.46 metros de altura.
```

É comum em um código a necessidade de comentar algo, para isso usamos o símbolo # e escrevemos o comentário após esse símbolo e portanto, tudo que é escrito na mesma linha após esse símbolo será ignorado na interpretação do código.

Podemos criar uma função em Python para executar um código a partir de parâmetros digitados pelo usuário. Para isso precisamos definir um nome para a função, os parâmetros que serão repassados pelo usuário e o corpo da função que contém uma sequência de códigos e que retornará um resultado solicitado.

No código abaixo, temos um exemplo de declaração de função em Python:

Código 1.4 – Declarando uma função em Python

```
1 def saudacao(nome,idade,cor):
2     print "Olá,",nome,"! Você tem",idade,"anos e gosta da cor",cor,"!"
3
4 saudacao("Maria Lúcia", 28, "vermelha")
```

A palavra reservada **def** indica que uma função será definida. No exemplo acima, essa função, cujo nome é **saudacao**, recebe três parâmetros ou argumentos, entre parênteses, um nome, uma idade e uma cor, e imprime a frase com os argumentos repassados na função. É importante notar o uso de dois pontos (:), após os argumentos entre parentêses, indicando que a partir dali começará o código da função. Observe que há uma indentação (recuo) no código para dizer que as linhas abaixo pertencem a função.

O resultado obtido da execução do código é dado a seguir:

```
Olá, Maria Lúcia ! Você tem 28 anos e gosta da cor vermelha !
```

Uma lista é um conjunto ordenado de valores, onde cada valor é identificado por um índice. Os valores que compõem uma lista são chamados elementos.

Usando listas, podemos adicionar elementos usando a função **.append()** como no exemplo que segue.

Código 1.5 – Adicionando elementos em uma lista em Python

```
1 notas = [5, 8, 9.5, 2]
2 notas.append(7)
3 print notas
```

```
[5, 8, 9.50000000000000, 2, 7]
```

Observe que o valor 7 foi adicionado na última posição da lista. Cada elemento é identificado com um índice ao lado do nome da lista, onde a primeira posição é a de índice 0. Por exemplo, usando a lista **notas** acima, temos **notas[0] = 5**, **notas[1] = 8** e **notas[4] = 7**.

Em Python, existem funções nativas como a **.append()** que vimos e funções que podem ser criadas com o **def**.

É muito comum o uso de listas que contém inteiros consecutivos e em Python há uma maneira simples de criá-los usando **range(a,b)**, em que a função **range** pega dois argumentos **a** e **b** e devolve uma lista que contém todos os inteiros de **a** até o **b**, incluindo o primeiro, mas não incluindo o último.

Código 1.6 – Listando números consecutivos em Python

```
1 range(2, 8)
```

```
[2, 3, 4, 5, 6, 7]
```

Se usarmos a função **range()** apenas com um argumento, como em **range(8)** ela cria uma lista que inicia em 0 e vai até o 7 (não inclui o argumento 8).

Código 1.7 – Listando números consecutivos de forma simples em Python

```
1 range(8)
```

```
[0, 1, 2, 3, 4, 5, 6, 7]
```

1.5.2 Operadores matemáticos

Como vimos na Seção 1.4, o SageMath pode ser usado como uma calculadora. Para isso, precisamos conhecer os operadores matemáticos.

Tabela 2 – Operadores Matemáticos em Python

Operadores	Operação	Exemplos
+	Adição	17 + 5 produz 22
-	Subtração	81 - 38 produz 43
*	Multiplicação	(5 - 2)*(1 + 7) produz 24
/	Divisão	(11 + 2)/(5 - 7) produz -13/2
//	Parte inteira da divisão	81//15 produz 5
%	Módulo ou resto da divisão	81%15 produz 6
^ ou **	Potenciação	11^2 ou 11**2 produz 121

Vamos usar os operadores no seguinte exemplo: imagine que queremos que o usuário digite dois números inteiros, um DIVIDENDO e um DIVISOR, e que seja retornado o Algoritmo da Divisão Euclidiana (DIVIDENDO = DIVISOR $\cdot$ QUOCIENTE + RESTO, em que o resto é menor que o divisor).

Já vimos anteriormente como criar uma função e dessa forma, vamos criar uma função onde o usuário entrará com os dois números inteiros, o DIVIDENDO e o DIVISOR, retornando o Algoritmo da Divisão Euclidiana.

Código 1.8 – Definindo a função que chamaremos de **euclidiana**.

```
1 def div_euclidiana(DIVIDENDO, DIVISOR):
```

Dados os valores solicitados, iremos descobrir o valor do RESTO e para isso, usaremos o operador matemático % da seguinte forma: RESTO = DIVIDENDO % DIVISOR.

Obtido o RESTO, podemos calcular o QUOCIENTE da seguinte forma: subtraindo o RESTO do DIVIDENDO, tornaremos a divisão pelo DIVISOR exata. Dessa forma, digitamos: QUOCIENTE = (DIVIDENDO - RESTO) / DIVISOR.

Outra forma mais simples de obter o QUOCIENTE é usando o operador que obtém a parte inteira da divisão (//), digitando: QUOCIENTE = DIVIDENDO // DIVISOR.

Por fim, retornaremos o Algoritmo da Divisão Euclidiana na forma DIVIDENDO = DIVISOR · QUOCIENTE + RESTO usando o comando **print** conforme código abaixo.

Código 1.9 – Função definida e imprimindo o Algoritmo.

```
1 def div_euclidiana(DIVIDENDO, DIVISOR):
2   RESTO = DIVIDENDO % DIVISOR
3   QUOCIENTE = DIVIDENDO // DIVISOR
4   print "Pelo Algoritmo da Divisão Euclidiana, tem-se: %d = %d x %d + %d
      ." %(DIVIDENDO,DIVISOR,QUOCIENTE,RESTO)
```

Usando a função definida acima digitando, por exemplo, **div_euclidiana(7,3)** e executando a função para esses valores, obtemos como resultado:

```
Pelo Algoritmo da Divisão Euclidiana, tem-se: 7 = 3 x 2 + 1.
```

1.5.3 Operadores relacionais e lógicos

Quando queremos comparar valores, usamos operadores relacionais ou também chamados operadores comparativos. Na comparação, obtemos duas possíveis respostas: **True** (Verdadeiro) ou **False** (Falso). São exemplos de operadores relacionais os que seguem na tabela a seguir.

Tabela 3 – Operadores Comparativos ou Relacionais em Python

Operadores Relacionais	Operação	Exemplos
>	maior que	print(15 > 87) produz False
<	menor que	print(27 < 51) produz True
>=	maior ou igual que	print(4.9 >= 1.3) produz True
<=	menor ou igual que	print(10 <= 9.9) produz False
==	igual a	print(10**2 == 100) produz True
<> ou !=	diferente de	print(2**4 != 4**2) produz False

Quando uma expressão é avaliada, só há duas possibilidades lógicas: ou ela é **verdadeira** ou é **falsa**; não se aceita uma terceira possibilidade. Essa ideia é chamada de **valor lógico** da expressão e constitui a base da computação: o 1(verdadeiro) ou 0 (falso), ligado ou desligado, aberto ou fechado.

Todo dado que assume esses dois possíveis valores (**True** ou **False**) é classificado como do tipo **booleano**.

Há três operadores lógicos que podem ser usado em uma expressão lógica: *and*, *or* e *not*. O operador *and*, usado entre duas ou mais expressões simples, resulta em **True**, se ambas as expressões simples também forem **True**. O operador *or*, usado entre duas ou mais expressões simples, resulta em **False**, se ambas as expressões simples também forem **False**. O operador *not*, usado para uma expressão, troca o valor lógico da expressão, resultando em **True**, se a expressão for **False**, e em **False**, se a expressão for **True**.

1.5.4 Estruturas de decisão

Sempre que precisamos tomar uma decisão quanto a executar ou não uma ação a partir de uma condição, usamos uma estrutura de decisão ou comumente chamada estrutura condicional. Para isso usamos uma estrutura formada pela palavra reservada **if**, seguida por uma condição e por dois pontos (:), conforme mostra o exemplo a seguir.

Código 1.10 – Usando a estrutura condicional IF em Python

```
1 def idade(x):
2   if x < 18:
3       print('Você é menor de idade!')
```

Observe que a condição para que seja executada a ação **print('Você é menor de idade!')** é verificar se a variável **idade** é menor que 18. Se a condição não for satisfeita, a ação citada não será executada, desconsiderando-a, e o códido segue normalmente após a estrutura do **if**.

Contudo, há situações em que quando a condição do **if** não é válida, deve-se executar outro código e para isso, usamos a palavra reservada **else**, seguida de dois pontos (:). Caso haja mais de uma condição alternativa que precisa ser verificada, devemos utilizar o **elif**, seguida por uma condição e por dois pontos (:), para avaliar as expressões intermediárias antes de usar o **else**, conforme veremos o exemplo a seguir.

Código 1.11 – Usando a estrutura condicional IF-ELIF-ELSE em Python

```
1 def idade(x):
2   if x < 16: print('Você ainda não pode votar!')
3   elif x >= 16 and x < 18:
4     print('Você pode votar, mas não é obrigatório!')
```

```
5    elif x >= 18 and x <= 70:
6      print('Você deve votar para mudar nosso país! O futuro do país está
       em suas mãos!')
7    else:
8      print('Você já contribuiu para o nosso país! Se desejar, ainda pode
       votar para mudar nosso país!')
```

1.5.5 Estruturas de repetição

É muito comum em um problema ter a necessidade de repetir um mesmo código diversas vezes enquanto uma condição dada é válida. Em situações como essa, usamos uma estrutura de repetição ou ainda conhecida por *loop* (laço de repetição).

Em Python, há duas formas de criar um laço de repetição: o **for** e o **while**.

O comando **for** percorre os itens de uma lista de dados ou coleção e, para cada um deles, executa um bloco de código; ao passo em que o comando **while**, executa um conjunto de instruções várias vezes enquanto uma condição é válida.

Vejamos exemplos dos uso desses comandos para entender a diferença entre eles:

Código 1.12 – Usando o FOR

```
1  def SomaTermosPA(a1,razao,n):
2      soma = 0
3      pa = []
4
5      for i in range(n):
6          b = a1 + i * razao
7          soma = soma + b
8          pa.append(b)
9
10     print "A soma dos",n,"termos
       da P.A.",pa,"é",soma
11
```

Código 1.13 – Usando o WHILE

```
1  def SomaTermosPA(a1,razao,n):
2      soma = 0
3      pa = []
4      i = 0
5      while i < n:
6          b = a1 + i * razao
7          soma = soma + b
8          pa.append(b)
9          i = i + 1
10     print "A soma dos",n,"termos
       da P.A.",pa,"é",soma
11
```

No código usando o comando **for** observamos que a variável **i** percorre os valores da lista **range(n)**, isto é, a variável **i** assume os valores de 0 até $n-1$, executando o bloco de código **n** vezes.

Já no comando **while** usamos, nesse exemplo, um contador (no código acima $\Rightarrow$ $\mathbf{i = i + 1}$) para fazer variar o valor de **i** que tem relação com a condição dada; caso contrário, entraremos em um laço infinito, ou seja, nunca atingiremos um fim na estrutura de repetição. Importante notar que poderíamos ter usado outra ideia para finalizar o comando **while** ao invés do contador, desde que em algum momento a condição se torne falsa para não cair em um laço infinito.

Usando a função SomaTermosPA nos dois casos (função com FOR e função com WHILE), obteremos os mesmos resultados como se observa ao executar SomaTermosPA(10, 3,25), que corresponde a soma dos 25 primeiros termos de uma PA cujo primeiro termo é 10 e a razão é 3.

◊ No primeiro código usando o comando **FOR**

```
A soma dos 25 termos da P.A. [10, 13, 16, 19, 22, 25, 28, 31, 34, 37,
40, 43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82] é 1150
```

◊ No segundo código usando o comando **WHILE**

```
A soma dos 25 termos da P.A. [10, 13, 16, 19, 22, 25, 28, 31, 34, 37,
40, 43, 46, 49, 52, 55, 58, 61, 64, 67, 70, 73, 76, 79, 82] é 1150.
```

2 NOÇÕES DE TEORIA DOS GRAFOS

Nesse capítulo, introduziremos os conceitos iniciais relacionados à Teoria dos Grafos, abordando algumas formas de representação de grafos no SageMath, sua representação matricial e a ideia de grau de um vértice, bem como outras definições que se constituem como básicas nessa área.

Alguns conceitos da Teoria dos Grafos foram suprimidos por serem usuais aos estudantes da graduação do curso de Licenciatura em Matemática e professores dessa área, e para uma abordagem mais completa o leitor pode consultar o livro *Graph Theory With Applications* de Bondy e Murty (1979) ou o livro *Grafos. Conceitos, Algoritmos e Aplicações.* de Marco Goldbarg e Elizabeth Goldbarg (2012).

2.1 PRIMEIRAS NOÇÕES

Exemplo 2 (Problema dos Amigos)**.** *Uma escola de idiomas está realizando uma viagem para uma praia em Pernambuco, com o objetivo de praticar a língua aprendida com os turistas que visitam essa praia. Com o intuito de não atingir apenas uma de suas unidades de ensino, essa escola de idiomas resolveu sortear alunos aleatoriamente dentre os matriculados em todas as unidades.*

Entre os premiados para realizarem essa viagem estão Ana, Bruno, Carla, Daniela, Eduardo, Fernanda e Gustavo. Nem todos dessa lista se conhecem e a escola propôs que os alunos que não tivessem contato uns com os outros também pudessem praticar entre si.

Para auxiliar nesse processo, a escola resolveu listar os nomes das pessoas que se conhecem, obtendo como resultado a lista a seguir:

- *Ana conhece Eduardo e Fernanda;*
- *Bruno conhece Carla, Daniela, Eduardo e Gustavo;*
- *Carla conhece Bruno, Fernanda e Gustavo;*
- *Daniela conhece Bruno e Gustavo;*
- *Eduardo conhece Ana e Bruno;*
- *Fernanda conhece Ana, Carla e Gustavo;*

- *Gustavo conhece Bruno, Carla, Daniela e Fernanda.*

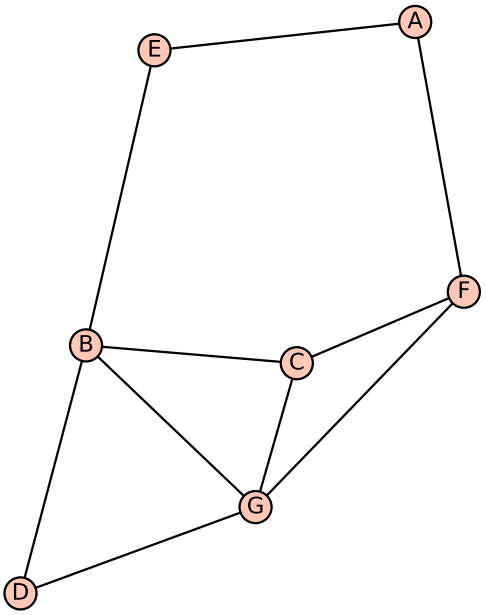
Mas será que nessa listagem não esquecemos de alguém? Será que existem dois grupos isolados, ou apenas um?

Para que possamos esclarecer essa possível dúvida, podemos representar essa mesma lista através de uma figura onde as pessoas são representadas por um vértice com a letra inicial de seus nomes e a informação de que duas pessoas X e Y se conhecem será representado por uma aresta ligando o vértice X ao vértice Y.

Essa estrutura, dotada da ideia de representar os objetos por vértices e, se houver alguma conexão entre esses objetos, estes serem ligados por arestas, constitui a ideia de **grafos**.

Essa estrutura citada acima pode variar seu formato conforme a figura escolhida, mas os vértices e suas conexões permanecerão as mesmas.

Figura 6 – Grafo do Problema dos Amigos



Com o auxílio dessa estrutura se torna mais fácil perceber que não esquecemos de nenhuma pessoa da nossa listagem, pois tal pessoa seria representado por um vértice isolado, e que existe um único grupo de pessoas, e não dois ou mais grupos. Assim a pessoa D pode pedir o contato da pessoa A através da pessoa B.

Agora que vimos um exemplo onde se pode utilizar a ideia de grafo, podemos escrevê-la mais formalmente:

- O Conjunto dos **Vértices** desse grafo é representado por V. No exemplo anterior, os vértices são as pessoas que foram premiadas com a viagem.

- O Conjunto das **Arestas** desse grafo é representado por A. No nosso exemplo, as arestas representam a ideia que as duas pessoas conectadas se conhecem.

Portanto, em um grafo qualquer, nos importa dizer quem são os vértices e quem são as arestas, ou seja, que pares de vértices estão conectados.

No problema da viagem realizada pela escola de idiomas (Exemplo 2), podemos representar o grafo usando a ideia de conjuntos, da seguinte forma:

$$V = \{\text{A, B, C, D, E, F, G}\},$$
$$A = \{\{\text{A,E}\}, \{\text{A,F}\}, \{\text{B,C}\}, \{\text{B,D}\}, \{\text{B,E}\}, \{\text{B,G}\}, \{\text{C,F}\}, \{\text{C,G}\}, \{\text{D,G}\}, \{\text{F,G}\}\} \text{ e}$$
$$\text{G} = (V,A)$$

Um grafo simples é um par ordenado $(V(G), A(G))$ que consiste em um conjunto não vazio $V(G)$ de *vértices* (ou *nós*) e um conjunto $A(G)$ de *arestas*, disjunto de $V(G)$. Cada aresta de G é um par não ordenado de *vértices* de G, distintos.

Se a é uma aresta e u e v são vértices tais que a aresta a **liga** o vértice u ao vértice v, dizemos que os vértices u e v são as *extremidades* de a.

Se dois vértices são ligados por uma aresta, dizemos que essa aresta é **incidente** aos vértices e que os vértices são **adjacentes** ou **vizinhos**.

Agora que já sabemos definir um grafo G como um objeto matemático, vamos mostrar como definir no SageMath:

Código 2.1 – Definindo o grafo no SageMath

```
1 V = ['A','B','C','D','E','F','G']
2 A = [['A','E'],['A','F'],['B','C'],['B','D'],['B','E'],['B','G'],['C','F
    '],['C','G'],['D','G'],['F','G']]
3
4 G = Graph([V,A])
```

Ou ainda:

Código 2.2 – Definindo o grafo pelas arestas no SageMath

```
1 A = [['A','E'],['A','F'],['B','C'],['B','D'],['B','E'],['B','G'],['C','F
    '],['C','G'],['D','G'],['F','G']]
2
3 G = Graph(A)
```

Outra forma de definir um grafo é usando uma lista que indica os vértices e seus vizinhos. Matematicamente, podemos escrever este grafo da seguinte forma:

$$\text{G} = \{\{\text{A,}\{\text{E, F}\}\}, \{\text{B,}\{\text{C, D, E, G}\}\}, \{\text{C,}\{\text{B, F, G}\}\}, \{\text{D,}\{\text{B, G}\}\}, \{\text{E,}\{\text{A, B}\}\},$$
$$\{\text{F, }\{\text{A, C, G}\}\}, \{\text{G, }\{\text{B, C, D, F}\}\}\}$$

No SageMath, a forma mais prática para definir um grafo também usa uma lista indicando os vértices e seus vizinhos, conforme segue.

Código 2.3 – Definindo o grafo de forma mais prática no SageMath

```
G = Graph({'A':['E','F'],'B':['C','D','E','G'],'C':['B','F','G'],'D':['B
    ','G'], 'E':['A','B'], 'F':['A','C','G'], 'G':['B','C','D','F']})
```

Podemos simplificar essa forma prática de definir o grafo, nesse caso não precisamos necessariamente repetir as arestas que já foram explicitadas, conforme é mostrado no código a seguir:

Código 2.4 – Definindo o mesmo grafo de forma mais simples no SageMath

```
G = Graph({'A':['E','F'],'B':['C','D','E','G'],'C':['F','G'],'D':['G'],
    'F':['G']})
```

Essa é a principal forma de definir um grafo no SageMath.

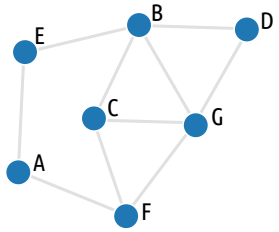
- Para visualizar e interagir com o grafo, modificando a posição dos vértices e alterando o formato do grafo, escrevemos o código **show(G)** e clicamos no botão "**run**" (ou **Shift** + **Enter**).

Código 2.5 – Grafo na Forma Interativa

```
G = Graph({'A':['E','F'], 'B':['C','D','E','G
    '], 'C':['F','G'], 'D':['G'], 'F':['G']})

show(G)     #usando a forma interativa

```

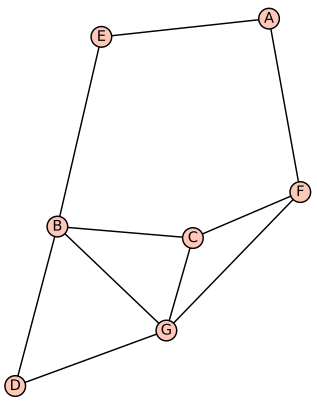


- Para visualizar de forma estática, escrevemos o código plot(**G**) e sempre que clicar no botão "**run**" (ou **Shift** + **Enter**) será gerado um novo formato para o grafo.

Código 2.6 – Grafo na Forma Estática

```
G = Graph({'A':['E','F'], 'B':['C','D','E','G
    '], 'C':['F','G'], 'D':['G'], 'F':['G']})

plot(G)     #usando a forma estática
```



2.2 GRAFOS

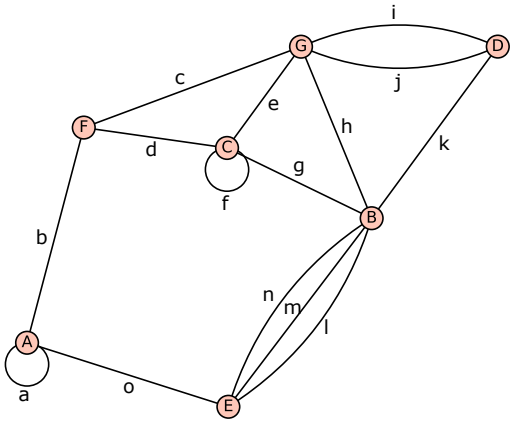
Como vimos anteriormente, um grafo simples G fica definido pelo conjunto não vazio de vértices $V(G)$ e pelo conjunto de arestas $A(G)$. Arbitrariamente, sempre que tivermos um grafo em discussão, este será nomeado de grafo **G**. Denotaremos $v(G)$ e $a(G)$ ao número de vértices e arestas de um grafo G, respectivamente.

Um **multigrafo** é um par ordenado $(V; A)$ de conjuntos disjuntos de vértices (V) e arestas (A), relacionados por uma **função de incidência** ψ em que toda aresta possui um ou dois vértices como extremidades. Se u e v são as extremidades de uma aresta a, escrevemos $\psi_G(a) = uv$.

Vejamos um exemplo de multigrafo e algumas definições importantes.

Exemplo 3. *Considere o grafo G mostrado a seguir e as funções de incidência ψ_G para cada aresta de G.*

Figura 7 – Grafo G



$$\psi(a) = AA \qquad \psi(i) = DG$$
$$\psi(b) = AF \qquad \psi(j) = DG$$
$$\psi(c) = FG \qquad \psi(k) = BD$$
$$\psi(d) = CF \qquad \psi(l) = BE$$
$$\psi(e) = CG \qquad \psi(m) = BE$$
$$\psi(f) = CC \qquad \psi(n) = BE$$
$$\psi(g) = BC \qquad \psi(o) = AE$$
$$\psi(h) = BG$$

No Exemplo 3, há uma aresta que liga o vértice A a ele mesmo. Chamamos de **laço** essa aresta cujas extremidades coincidem com o mesmo vértice. É possível observar outro laço no vértice C.

Fixado um par de vértices $\{A, B\}$, se duas ou mais arestas incidem sobre A e B, estas arestas são ditas **arestas paralelas**. Podemos observar arestas paralelas conectando os vértices B e E e conectando os vértices D e G.

Podemos dizer que um grafo simples é um multigrafo que não possui laços ou arestas paralelas. Por exemplo, o grafo da Figura 7 acima *não* é simples, pois contém laços e/ou arestas paralelas.

Ao declarar um multigrafo, precisamos mostrar a função de incidência para todas as arestas, pois no caso em que há arestas paralelas precisamos distingui-las. Já no caso de grafos simples essa função pode ser omitida.

No SageMath, para saber se um grafo G tem laço (em inglês, *has loops*), escrevemos **G.has_loops()**.

No SageMath, podemos obter todas as arestas de um grafo G existente usando **G.edges()**. Cada aresta é apresentado como uma tripla (u, v, l), onde u e v são os vértices conectados por essa aresta e l é o rótulo dessa aresta. Para acessar cada termo, usamos o índice corresponte, sendo 0 para o primeiro vértice, 1 para o segundo vértice e 2 para o rótulo.

No SageMath também é possível obter uma lista com todos os laços existentes em um grafo G digitando **G.loops()**. Por exemplo, usando o grafo G do Exemplo 3, obtemos o seguinte resultado:

```
[('A', 'A', None), ('C', 'C', None)]
```

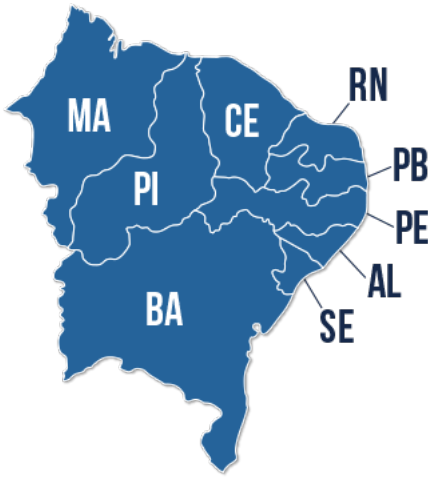
Podemos ainda excluir todos os laços do grafo G usando a função **remove_loops()**.

De forma semelhante ao caso dos laços, temos as funções referentes a arestas paralelas (*multiple edges*):

- **G.has_multiple_edges()** para saber se um grafo G tem arestas paralelas;
- **G.multiple_edges()** para listar todas as arestas paralelas; e
- **G.remove_multiple_edges()** para excluir todas as arestas paralelas do grafo G.

Exemplo 4 (Fronteiras do Nordeste Brasileiro)**.** *São nove os Estados do Nordeste do Brasil. Observando o mapa do Nordeste brasileiro na Figura 8 a seguir, podemos notar que alguns estados têm fronteiras com outros.*

Figura 8 – Mapa da Região Nordeste



Disponível em:
<*http://www.blukit.com.br/frontend/images-map/mapa_regiao_nordeste.png*>

Desejamos representar tal situação como um grafo tendo os estados como vértices e conectamos esses estados por arestas se eles tiverem alguma fronteira. Vamos plotar o grafo acima no SageMath:

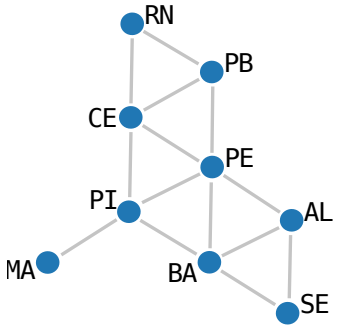
Código 2.7 – Problema do mapa do Nordeste no SageMath

```
G = Graph({'AL': ['BA','PE','SE'],'BA': ['PE','PI','SE'], 'CE': ['PB','
    PE','PI','RN', 'MA': ['PI'], 'PB': ['PE','RN'],'PE': ['PI']})

show(G)
```

Executando o código, obtemos o grafo dado pela Figura 9 abaixo:

Figura 9 – Grafo das fronteiras dos estados da Região Nordeste



Observe que esse grafo do Exemplo 4 é simples, pois não apresenta laços, nem arestas paralelas.

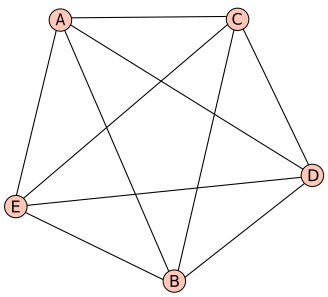
Chamamos de **ordem** de um grafo G ao número de vértices de G, isto é, $v(G)$. Chamamos de **tamanho** de um grafo G ao número de arestas de G, isto é, $a(G)$.

Por exemplo, no grafo do Exemplo 4, nomeamos esse grafo de G, então $v(G) = 9$ e $a(G) = 14$, isto é, esse grafo tem ordem 9 e tamanho 14.

Para obter a ordem (em inglês, *order*) de um grafo G, digitamos **G.order()** ou **len(G)**, ou ainda **len(G.vertices())**. Para obter o tamanho (em inglês, *size*) de um grafo G, digitamos **G.size()** ou **len(G.edges())**.

Exemplo 5. *Considere o grafo G a seguir:*

Figura 10 – Grafo G



Um grafo simples em que cada par de vértices distintos é ligado por uma aresta é dito um **grafo completo**. Representamos um grafo completo com n vértices por K_n.

No Exemplo 5, temos um grafo completo K_5, pois cada vértice está conectado a todos os outros por uma aresta.

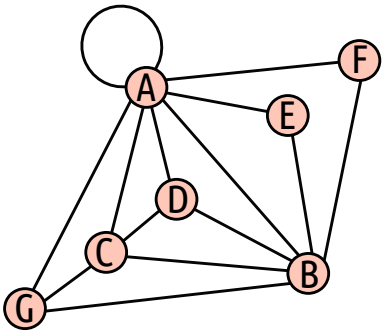
Podemos obter um grafo completo com n vértices no SageMath, digitando o comando **graphs.CompleteGraph(n)**. Por exemplo, o grafo do Exemplo 5 pode ser obtido no SageMath digitando **G = graphs.CompleteGraph(5)** e visualizando com um **plot(G)**.

2.3 REPRESENTANDO UM GRAFO POR UMA MATRIZ

Podemos representar um grafo G qualquer por meio de uma matriz, cujas linhas representam os vértices e as colunas representam as arestas, ou seja, uma matriz do tipo $v(G) \times a(G)$.

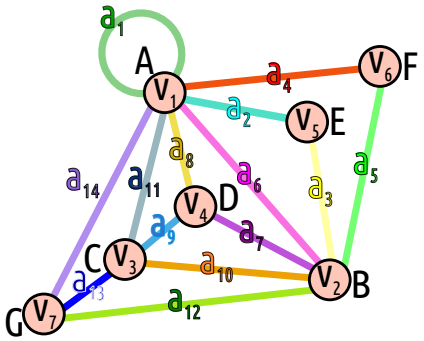
Exemplo 6. *Considere o grafo G a seguir:*

Figura 11 – Grafo G



Nomearemos os vértices e arestas para entender a construção da matriz relativa ao grafo G.

Figura 12 – Grafo G com vértices e arestas renomeados.



Nesse exemplo, temos 7 vértices e 14 arestas. Portanto, a matriz que será construída será do tipo 7 x 14.

A matriz M(G), chamada **matriz de incidência de G**, será a matriz formada pelos elementos m_{ij} cujos valores indicam o número de vezes (0, 1 ou 2) em que o vértice v_i é incidente com a aresta a_j. Melhor dizendo, terá valor 0 se v_i e a_j não forem incidentes; terá valor 1 se v_i e a_j forem incidentes; e valor 2 se a_j for um laço com extremos em v_i.

Portanto, a matriz de incidência do grafo da Figura 12 é dada por:

Tabela 4 – Matriz de incidência do grafo G - M(G)

	a_1	a_2	a_3	a_4	a_5	a_6	a_7	a_8	a_9	a_{10}	a_{11}	a_{12}	a_{13}	a_{14}
v_1	2	1	0	1	0	1	0	1	0	0	1	0	0	1
v_2	0	0	1	0	1	1	1	0	0	1	0	1	0	0
v_3	0	0	0	0	0	0	0	0	1	1	1	0	1	0
v_4	0	0	0	0	0	0	1	1	1	0	0	0	0	0
v_5	0	1	1	0	0	0	0	0	0	0	0	0	0	0
v_6	0	0	0	1	1	0	0	0	0	0	0	0	0	0
v_7	0	0	0	0	0	0	0	0	0	1	0	1	1	1

A ideia de usar a representação de um grafo por meio de uma matriz pode facilitar o entendimento de um problema. Curiosamente, a soma dos valores de qualquer coluna em uma matriz de incidência é sempre 2. Notou isso? E a explicação, será que você consegue perceber o porquê? A justificativa é que cada aresta incide sobre dois vértices.

Há também uma outra matriz A(G), chamada **matriz de adjacência do grafo G**, cuja ordem dessa matriz é igual ao número de vértices do grafo, isto é, uma matriz do tipo $v(G) \times v(G)$. Seus elementos indicam o número de arestas que ligam v_i a v_j.

No Exemplo 6, a matriz de adjacência do grafo G é dado por:

Tabela 5 – Matriz de adjacência do grafo G - A(G)

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	1	1	1	1	1	1	1
v_2	1	0	1	1	1	1	1
v_3	1	1	0	1	0	0	1
v_4	1	1	1	0	0	0	0
v_5	1	1	0	0	0	0	0
v_6	1	1	0	0	0	0	0
v_7	1	1	1	0	0	0	0

Vale ressaltar que a matriz de adjacência de um grafo G é uma matriz simétrica, ou seja, $a_{ij} = a_{ji}$. Uma outra reflexão: O que a soma das colunas ou a soma das linhas de uma matriz de adjacência de um grafo G representa? Ela indica o número de vértices que um vértice está conectado, o qual chamamos de **grau desse vértice**.

Antes de conhecermos a forma de obter as matrizes de incidência e de adjacência de um grafo G usando o SageMath, precisamos conhecer como o SageMath define a ordem dos vértices e das arestas que ocuparão as linhas e/ou colunas em cada tipo das matrizes citadas.

Para isso, há um comando para saber a ordem dos vértices definida pelo SageMath em relação a um grafo G dada por **G.vertices()**. Já o comando para saber a ordem das arestas definida pelo SageMath, digitamos o comando **G.edge_iterator()**.

No caso do Exemplo 6 apresentado anteriormente, se digitarmos os códigos acima, obtemos os vértices e arestas ordenados pelo SageMath:

Código 2.8 – Reconhecendo a ordenação dos vértices e arestas feita pelo SageMath

```
G = Graph({'A': ['A','B','C','D','E','F','G'], 'B': ['C','D','E','F','G
    '], 'C': ['D','G']})

G.vertices()
for i in G.edge_iterator(): print i
```

Ao executar o código, obtemos o seguinte resultado:

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
('A', 'A', None)
('A', 'C', None)
('A', 'B', None)
('A', 'E', None)
('A', 'D', None)
('A', 'G', None)
('A', 'F', None)
('B', 'C', None)
('C', 'D', None)
('C', 'G', None)
('B', 'E', None)
('B', 'D', None)
('B', 'G', None)
('B', 'F', None)
```

Portanto, as linhas da matriz de incidência e as linhas e colunas da matriz de adjacência de um grafo G serão ocupadas pelos vértices seguindo a ordenação: (1ª) 'A'; (2ª) 'B'; (3ª) 'C'; (4ª) 'D'; (5ª) 'E'; (6ª) 'F'; e (7ª) 'G'; assim como as colunas da matriz de incidência desse mesmo grafo ocupadas pelas arestas seguiria a seguinte ordem: (1ª)

(‘A’, ‘A’, None); (2ª) (‘A’, ‘C’, None); (3ª) (‘A’, ‘B’, None); (4ª) (‘A’, ‘E’, None); (5ª) (‘A’, ‘D’, None); (6ª) (‘A’, ‘G’, None); (7ª) (‘A’, ‘F’, None); (8ª) (‘B’, ‘C’, None); (9ª) (‘C’, ‘D’, None); (10ª) (‘C’, ‘G’, None); (11ª) (‘B’, ‘E’, None); (12ª) (‘B’, ‘D’, None); (13ª) (‘B’, ‘G’, None); e (14ª) (‘B’, ‘F’, None).

Perceba que todas as arestas são representadas pelos seus extremos e a palavra “*None*” (o que se traduz como “Nenhum”) seria um nome ou rótulo para essa aresta, que nesse caso não há rótulos para as arestas citadas, mas não precisamos nos preocupar com tal detalhe.

Para obter a matriz de incidência de um grafo G, basta digitar G.incidence_matrix() e o SageMath retornará a matriz solicitada seguindo a ordenação dos vértices e arestas conforme citamos acima. Ao executarmos tal código, obtemos o resultado a seguir:

```
[1 1 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 0 1 1 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 1 0]
[0 0 1 0 0 0 0 1 0 1 1 1 1 2]
[1 0 0 1 1 1 1 1 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 1 0 0 1 0 0]
```

Para obter a matriz de adjacência de um grafo G, digitamos G.adjacency_matrix() e o SageMath retornará a matriz solicitada seguindo a ordenação dos vértices nas linhas e colunas conforme citamos acima. Ao executarmos tal código, obtemos o resultado a seguir:

```
[0 0 1 0 1 1 0]
[0 0 0 0 1 1 0]
[1 0 0 0 1 1 1]
[0 0 0 0 1 1 0]
[1 1 1 1 1 1 1]
[1 1 1 1 1 0 1]
[0 0 1 0 1 1 0]
```

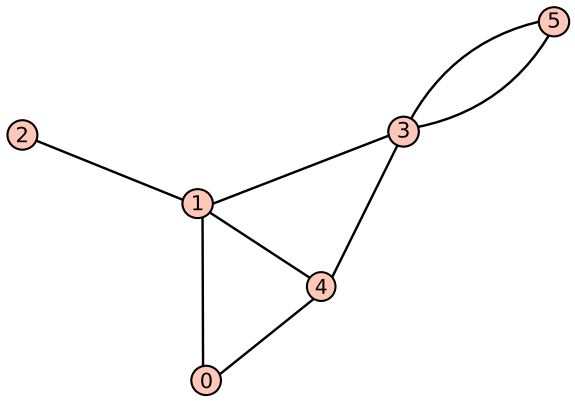
Por outro lado, a partir de uma matriz de incidência ou de uma matriz de adjacência podemos obter o grafo relacionado a essa matriz. Para exemplificar esse fato, vamos considerar a matriz de incidência M a seguir e imprimir o grafo correspondente a M.

Código 2.9 – Obtendo o grafo a partir da matriz de incidência no SageMath

```
1 M = Matrix([[0,0,0,1,0,1,0,0], [0,0,0,0,1,1,1,1], [0,0,0,0,0,0,0,1],
     [1,1,1,0,1,0,0,0], [0,0,1,1,0,0,1,0], [1,1,0,0,0,0,0,0]])
2
```

```
3 H = Graph(M)
4 plot(H)
```

Ao executar o código, obtemos o seguinte resultado:

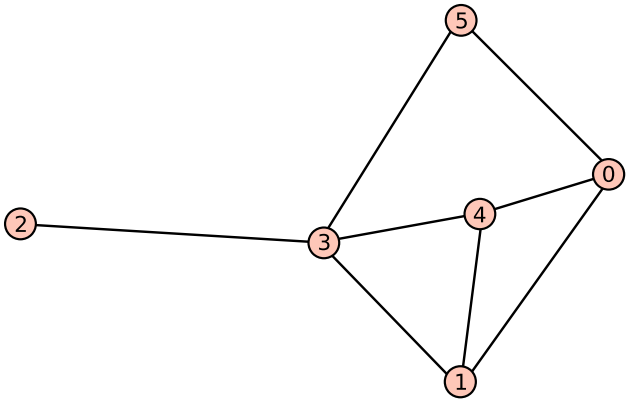


Consideramos agora a matriz de adjacência A a seguir e imprimimos o grafo correspondente a A, conforme código abaixo.

Código 2.10 – Obtendo o grafo a partir da matriz de adjacência no SageMath

```
1 A = Matrix([[0,1,0,0,1,1], [1,0,0,1,1,0], [0,0,0,1,0,0], [0,1,1,0,1,1],
     [1,1,0,1,0,0], [1,0,0,1,0,0]])
2
3 H = Graph(A)
4 plot(H)
```

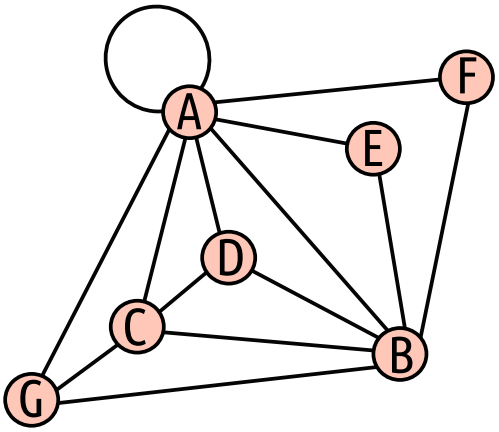
Ao executar o código, obtemos o seguinte resultado:



2.4 GRAU DE UM VÉRTICE

O grau $d(v)$ de um vértice v de um grafo G é dado pelo número de arestas que incidem sobre o vértice v, com os laços contados duas vezes.

Por exemplo, observando o grafo G do Exemplo 6, temos os seguintes graus de vértices:



- A → grau 8
- B → grau 6
- C → grau 4
- D → grau 3
- E → grau 2
- F → grau 2
- G → grau 3

Denotaremos o **grau mínimo** por $\delta(G)$ e o **grau máximo** por $\Delta(G)$ dentre os graus dos vétices do grafo **G**. Analisando o Exemplo anterior, temos $\delta(G) = 2$ e $\Delta(G) = 8$.

Se somarmos todos os graus do grafo acima, temos como resultado 28, um número par.

Os graus dos vértices no grafo do problema dos amigos (Exemplo 2), temos {2, 4, 3, 2, 2, 3, 4}, cuja soma dos graus de cada vértice resulta em 20, um número par.

No Exemplo 4, sobre o Nordeste Brasileiro, os graus dos vértices são {3, 4, 4, 1, 3, 5, 4, 2, 2}, cuja soma dos graus dá 28, também um número par.

Mas será que a soma dos graus dos vértices de um grafo é sempre par? A resposta é **SIM** e essa ideia constitui o nosso próximo teorema.

Teorema 7. *A soma dos graus de todos os vértices v de um grafo G qualquer é sempre par e igual ao dobro do número de arestas $|A(G)|$ de G. Ou seja,*

$$\sum_{v \in V(G)} d(v) = 2\,|A(G)|$$

Demonstração: Cada aresta de G contribui com 2 graus para a soma dos graus de todos os vértices. Assim, a soma é igual a duas vezes o número de arestas de G. ∎

Uma consequência imediata desse Teorema 7 é o corolário a seguir:

Corolário 8. *Em qualquer grafo, o número de vértices de grau ímpar é par.*

Demonstração: Vamos inicialmente separar os vértices $V(G)$ em dois conjuntos: os de grau ímpar em um conjunto V_1 e os de grau par no conjunto V_2. Aqui omitimos o (G) em $V_1(G)$ e $V_2(G)$ para simplificar a escrita para V_1 e V_2.

Então o resultado do Teorema 7 nos indica que $\sum_{v \in V(G)} d(v) = 2\,|A(G)|$, mas como particionamos o conjunto $V(G)$ nos conjuntos V_1 e V_2, podemos escrever:

$$\sum_{v \in V_1} d(v) + \sum_{v \in V_2} d(v) = 2\,|A(G)|\,.$$

Como $\sum_{v \in V_2} d(v)$ é par, já que é um somatório de graus pares; e o termo $2\,|A(G)|$ também é par, então concluímos que $\sum_{v \in V_1} d(v)$ também é par, já que é uma diferença de números pares, a saber $\sum_{v \in V_1} d(v) = 2\,|A(G)| - \sum_{v \in V_2} d(v)$.

Portanto, para que $\sum_{v \in V_1} d(v)$ seja par, temos que ter um número par de vértices de grau ímpar, ou seja, $|V_1|$ é par, o que prova o colorário. ∎

Um grafo G é dito $k-$*regular* se $d(v) = k$ para todo $v \in V(G)$. Um grafo é **regular** se for $k-$ regular para algum k. No Exemplo 5, cada vértice teria grau 4, o que o torna um grafo 4-regular. Generalizando, grafos completos (Exemplo 5) são regulares.

Para saber o grau (em inglês, *degree*) de um vértice **v** de um grafo **G** pelo SageMath, escrevemos **G.degree(v)**.

Para obter a sequência de graus de cada vértice de **G**, de acordo com a ordenação dos vértices feita pelo SageMath (que nem sempre se mantém igual a ordem obtida pelo código **G.vertices()**), basta digitar **G.degree(labels=true)**, onde **labels=true** indica que mostrará o rótulo ou nome de cada vértice do gráfico seguido de seus respectivos graus.

Retomando o Exemplo 2 sobre o problema dos amigos e digitando o código citado **G.degree(labels=true)**, obtemos o seguinte resultado:

```
'A': 2, 'C': 3, 'B': 4, 'E': 2, 'D': 2, 'G': 4, 'F': 3
```

Observe que o SageMath trocou a ordem dos vértices colocando o 'C' antes do 'B' e o 'E' antes do 'D', outra opção é citar os vértices explicitamente como **G.degree(vertices= ['A','B','C','D','E','F','G'])**, obtendo:

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

Se um grafo **G** é simples, isto é, não possui laços nem arestas paralelas, como no Exemplo 2, é interessante notar o significado dos valores que pertencem a diagonal principal da matriz $\mathbf{A^2}$, em que **A** é a matriz de adjacência do grafo **G**.

Vamos calcular o resultado da potência **2** da matriz de adjacência **A** relativa ao grafo **G** do problema dos amigos (Exemplo 2) obtida através do SageMath.

$$\begin{pmatrix} 2 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 4 & 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 3 & 2 & 1 & 1 & 2 \\ 0 & 1 & 2 & 2 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 & 1 & 3 & 1 \\ 1 & 2 & 2 & 1 & 1 & 1 & 4 \end{pmatrix}$$

Associou os valores dessa diagonal principal a algo? Elas coincidem com os graus de cada vértice de **G**, de acordo com a ordenação padrão do SageMath (**G.vertices()**).

Esse resultado pode ser provado e será tema do exercício proposto ao aluno no Capítulo 5 e sua demonstração será apresentada na solução do exercício, encontrada no mesmo capítulo.

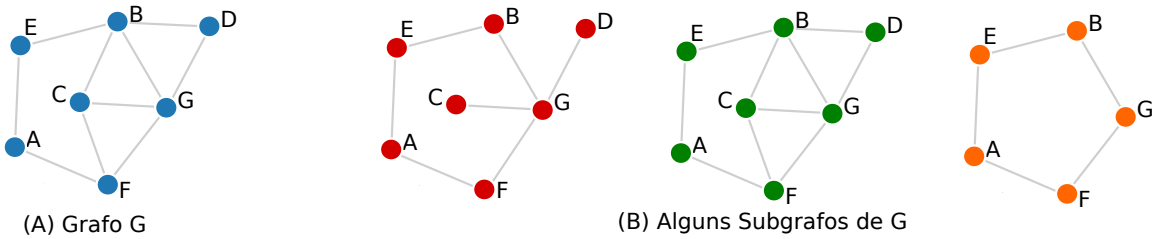
2.5 SUBGRAFOS

Um grafo H é **subgrafo** de G e escrevemos $H \subseteq G$, se $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$, ou seja, se todo vértice de H é vértice de G e toda aresta de H é aresta de G.

Se **H** é um **subgrafo** de **G**, então **G** é um **supergrafo** de **H**.

Para exemplificar essa ideia, vamos relembrar o grafo do problema dos amigos (Exemplo 2) e mostrar alguns subgrafos dele.

Figura 13 – Exemplos de Subgrafos



Um subgrafo H de um grafo G é dito **próprio** se H for diferente de G, ou seja, se H tiver, pelo menos, uma aresta ou vértice a menos que G.

Por exemplo, os dois primeiros subgrafos na Figura 13 - (B) acima são exemplos de **subgrafos próprios de G**.

Um subgrafo de um grafo G é dito ser um **subgrafo gerador** se ele contém todos os vértices de G, como pode se observar nos dois primeiros subgrafos na Figura 13 - (B) acima.

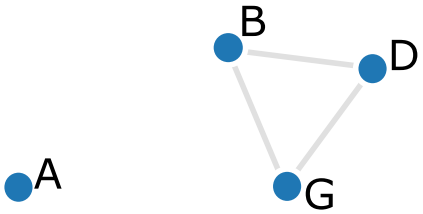
Um subgrafo H de um grafo G é dito **induzido** se toda aresta de G que tem extremos em H é também aresta de H. O último subgrafo da Figura 13 - (B) acima é um exemplo de subgrafo induzido de G (Figura 13 - [A]).

Dado um conjunto X de vértices de um grafo G, dizemos que um subgrafo de G é **induzido por X**, representado por $G[X]$, se esse subgrafo é formado por elementos de X e por todas arestas de G que têm os dois extremos em X.

Para obter um subgrafo contendo determinados vértices no SageMath, escrevemos **H = G. subgraph([u, v, w])**, em que **u**, **v** e **w** são os vértices de G e que formarão H juntos com as arestas que ligam esses vértices.

No grafo do problema dos amigos (Exemplo 2), vamos encontrar o subgrafo que usa os vértices A, B, D e G, digitando o código **H = G. subgraph(['A', 'B', 'D', 'G'])** e obtendo a figura a seguir.

Figura 14 – Subgrafo de G contendo os vértices A, B, D e G

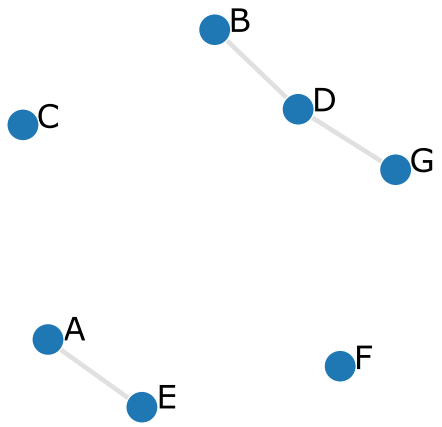


Para obter um subgrafo contendo determinadas arestas no SageMath, escrevemos **H = G. subgraph(edges = [(u, v), (v, w), (u,w)])**, em que **u**, **v** e **w** são os vértices

de **G** e **(u,v)**, **(v,w)** e **(u,w)** são as arestas de **G** que formarão **H** juntos todos os outros vértices que compõem **G**.

No grafo do problema dos amigos (Exemplo 2), podemos encontrar o subgrafo que contém as arestas {'A','E'}, {'B','D'}, {'D','G'} e {'E','C'}, digitando **H = G.subgraph(edges = [('A','E'), ('B','D'), ('D','G'), ('E','C')])**, obtendo o subgrafo que segue.

Figura 15 – Subgrafo de G contendo as arestas {'A','E'}, {'B','D'}, {'D','G'} e {'E','C'}



3 CAMINHOS E GRAFOS CONEXOS

Um **passeio em um grafo G** é uma sequência, finita e não vazia, de vértices e arestas de G, alternadamente, $(v_0, a_1, v_1, ..., a_{v-1}, v_{i-1}, a_i, v_i, a_{i+1}, v_{i+1}, ..., v_{k-1}, a_k, v_k)$, para todo $0 \leq i \leq k-1$, tal que v_i e v_{i+1} são vértices adjacentes, ou seja, são ligados por uma aresta. Chamamos os vértices v_0 e v_k, respectivamente, de **origem** e **fim do passeio**, enquanto que os vértices $v_1, ..., v_{k-1}$ são chamados **vértices internos ao passeio**. O inteiro **k** é o **comprimento do passeio**.

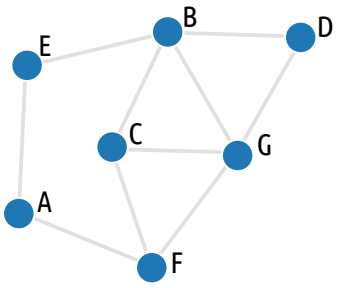
Observe que nessa definição de passeio pode haver repetição de vértices e arestas e pode ser aberto (quando a origem do passeio é diferente do fim do passeio) ou fechado (caso a origem e o fim do passeio coincidem). Em grafos simples, um passeio $P = (v_0, a_1, v_1, ..., v_{k-1}, a_k, v_k)$ é determinado pela sequência $(v_0, v_1, ..., v_{k-1}, v_k)$ de vértices, já que não há arestas paralelas nem laços pela definição. Se P é um passeio de v_0 a v_k, dizemos que P é um passeio-(v_0, v_k).

Se $P = (v_0 a_1 v_1 \dots a_k v_k)$ e $P' = (v_k a_{k+1} v_{k+1} \dots a_j v_j)$ são passeios em um grafo G, definimos o passeio $(v_k a_k v_{k-1} \dots a_1 v_0)$ como sendo o **reverso de P** e denotamos por P^{-1}; e o passeio $(v_0 a_1 v_1 \dots a_k v_k a_{k+1} \dots a_j v_{j)}$ como uma **concatenação de** P **e** P', denotada PP'. Uma **seção** de um passeio $P = (v_0 a_1 v_1 \dots a_k v_k)$ é um passeio que é uma subsequência de $(v_i a_{i+1} v_{i+1} \dots a_m v_m)$ de P e denotamos seção-(v_i, v_m) de P.

Se as arestas de um passeio são distintas, então chamamos esse passeio de **trilha**, e nesse caso, o comprimento da trilha é exatamente o número de arestas existente nela.

Se uma trilha tiver vértices distintos, então ela é chamada de **caminho**. Ou seja, um **caminho** é passeio com vértices e arestas distintas.

No problema dos amigos (Exemplo 2), podemos exemplificar alguns passeios, trilhas e caminhos, conforme abaixo. Como se trata de um grafo simples, iremos apenas escrever a sequência dos vértices, já que não há arestas paralelas. Caso não fosse um grafo simples, deveríamos especificar as arestas que fazem parte do passeio, trilha ou caminho.



- passeio: E - B - C - G - B - E - A - F - G - C
- trilha: E - B - C - G - B - D - G - F
- caminho: E - B - C - G - F - A

- passeio: F - C - G - D - B - G - F
- trilha: F - C - G - D - B - G - F
- caminho: F - C - B - D - G

No SageMath, é possível listar todos os passeios existentes entre dois vértices distintos v_i e v_j, sem que haja repetições de vértices e arestas, ou seja, listar os **caminhos** (em inglês, *paths*) de um grafo G.

Então, para listar todos os caminhos (em inglês, *all paths*) existentes entre o vértice de partida u e o vértice de chegada v de um grafo **G**, basta usar o código **G. all_paths(u, v)**.

Se desejarmos encontrar todos os caminhos entre os vértices **'A'** e **'D'** do **G** do problema dos amigos (Exemplo 2), digitamos **G. all_paths('A', 'D')** e obtemos como resultado:

```
[['A', 'E', 'B', 'C', 'G', 'D'], ['A', 'E', 'B', 'C', 'F', 'G',
'D'], ['A', 'E', 'B', 'D'], ['A', 'E', 'B', 'G', 'D'], ['A',
'F', 'C', 'B', 'D'], ['A', 'F', 'C', 'B', 'G', 'D'], ['A', 'F',
'C', 'G', 'B', 'D'], ['A', 'F', 'C', 'G', 'D'], ['A', 'F', 'G',
'C', 'B', 'D'], ['A', 'F', 'G', 'B', 'D'], ['A', 'F', 'G', 'D']]
```

Escrevendo **show(G. all_paths('A', 'D'))**, obtemos o resultado acima de uma forma mais limpa:

```
[[A, E, B, C, G, D], [A, E, B, C, F, G, D], [A, E, B, D], [A,
E, B, G, D], [A, F, C, B, D], [A, F, C, B, G, D], [A, F, C,
G, B, D], [A, F, C, G, D], [A, F, G, C, B, D], [A, F, G, B,
D], [A, F, G, D]]
```

Se quisermos saber qual o menor caminho (em inglês, *shortest path*) entre dois vértices, digitamos o código **G.shortest_path(u,v)**.

Por exemplo, se digitarmos **G.shortest_path('A','D')**, descobrimos o menor caminho entre os vértices 'A' e 'D' dado por:

```
['A', 'E', 'B', 'D']
```

É possível também obter todos os menores caminhos partindo de um vértice **v** de um grafo **G** através do código: **G.shortest_paths(v)**.

Por exemplo, se executarmos o código **G.shortest_paths('A')** no problema acima, obtemos os menores caminhos partindo do vértice 'A' cujos resultados estão listados abaixo:

```
'A': ['A'], 'C': ['A', 'F', 'C'], 'B': ['A', 'E', 'B'], 'E':
['A', 'E'], 'D': ['A', 'F', 'G', 'D'], 'G': ['A', 'F', 'G'],
'F': ['A', 'F']
```

Para obter os comprimentos dos menores caminhos (em inglês, *shortest path lengths*) partindo de um vértice **v** de um grafo **G**, escrevemos o código: **G.shortest_path_lengths(v)**.

Portanto, digitando **G. shortest_path_lengths('A')**, obtemos os seguintes resultados conforme podemos conferir com os resultados acima:

```
'A': 0, 'C': 2, 'B': 2, 'E': 1, 'D': 3, 'G': 2, 'F': 1
```

Um caminho ligando o vértice u ao vértice v é representado por **caminho-(u,v)**.

Dois vértices u e v de um grafo G estão **conectados** se existe um **caminho-(u,v)** em G. Em geral, definimos uma relação binária no conjunto dos vértices de um grafo G da seguinte forma: sejam u e v vértices de G, $u \sim v$ se e somente se u e v estão conectados.

Essa relação é de equivalência, pois convencionalmente v está conectado a v, para todo $v \in V(G)$ (propriedade reflexiva); se v está conectado a u em G, então u está conectado a v em G (propriedade simétrica); e se v está conectado a u e u está conectado a w, então v está conectado a w (propriedade transitiva).

Segue daí que o conjunto dos vértices de G fica particionado em algumas classes de equivalência, digamos V_1, V_2, ..., V_k. Note que qualquer aresta de G é incidente a vértices pertencentes a uma única dessas classes de equivalência.

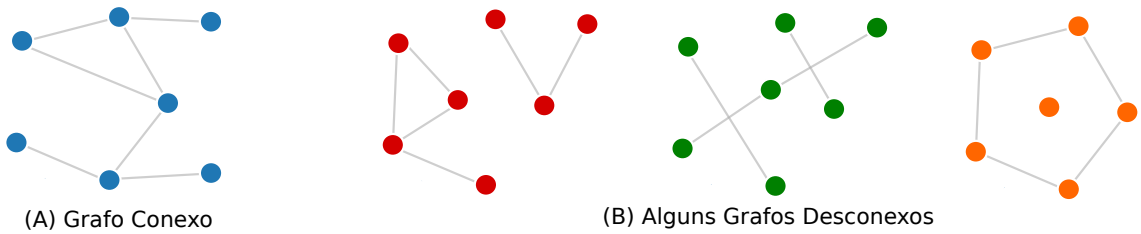
Logo, existe uma partição do conjunto de arestas de G, digamos A_1, A_2, ..., A_k, onde A_i é o conjunto de arestas de G que são incidentes apenas a vértices de V_i.

Seja G_i o grafo que possui o conjunto de vértices V_i e o conjunto de arestas A_i, submetidos a mesma função de incidência ψ de G. Dizemos que G_1, G_2, ... G_k são as **componentes conexas** desse grafo G.

Se um grafo G possui uma única classe de equivalência, é dito **grafo conexo**. Caso G possua mais de uma classe de equivalência, o grafo G é dito **desconexo**. Cada componente conexa de G é um grafo conexo e representamos o número de componentes conexas de um grafo G por $\omega(G)$.

A figura a seguir mostra um exemplo de um grafo conexo e outros três desconexos.

Figura 16 – Exemplos de grafos conexos e desconexos



O número de componentes conexas dos grafos desconexos são, respectivamente, $\omega(G) = 2$, $\omega(G) = 3$ e $\omega(G) = 2$.

No SageMath, para saber se o grafo G é conexo, digitamos **G.is_connected()**, que retornará o valor *true* ou *false*, e para obter o número de componentes conexas de G, digitamos **G.connected_components_numbers()**.

Uma análise interessante da matriz de adjacência de um grafo G **[A(G)]**(Seção 2.3) e sua relação com passeios gira em torno de uma **potência k dessa matriz**, isto é, $A^k = \underbrace{A \times \ldots \times A}_{k \text{ vezes}}$.

Teorema 9 (Potência k da Matriz de Incidência de um Grafo G)**.** *Se A é a matriz de adjacência de um grafo G com conjunto de vértices dado por $V(G) = \{v_1, v_2, ..., v_n\}$, então a entrada (i, j) de A^k, com $k \geq 1$, corresponde ao número de passeios (distintos) de comprimento k existentes entre os vértices v_i e v_j.*

Prova por Indução: Para $k = 1$, o resultado coincide com a ideia de matriz de adjacência A de um grafo, já que existe um passeio de comprimento 1 entre os vértices v_i e v_j, se e somente se, os vértices v_i e v_j são adjacentes, ou seja, existe uma aresta de G conectando esses vértices.

Seja $A^{k-1} = \left[a_{ij}^{(k-1)}\right]$ e assuma que $a_{ij}^{(k-1)}$ é o número de passeios diferentes de comprimento $k-1$ entre os vértices v_i e v_j no grafo G.

Queremos provar que cada entrada (i, j) de A^k equivale ao número de passeios distintos de comprimento k existentes entre os vértices v_i e v_j.

Consideremos $A^k = \left[a_{ij}^{(k)}\right]$. Como $A^k = A^{k-1} \cdot A$, temos que os elementos $a_{ij}^{(k)}$ dessa matriz são tais que $a_{ij}^{(k)} = \sum_{p=1}^{n} a_{ip}^{(k-1)} \cdot a_{pj}$.

Observe que o elemento $a_{ij}^{(k)}$, pela expressão acima, é obtido multiplicando-se os elementos da linha i de A^{k-1} pelos respectivos elementos da coluna j de A e, na sequência, efetuando-se a soma dos produtos obtidos.

Dessa forma, todo passeio entre v_i e v_j de comprimento k no grafo G consiste de um passeio entre v_i e v_p de comprimento $k-1$, onde o vértice v_p é adjacente a v_j, seguido da aresta que liga v_p a v_j e do vértice v_j.

Portanto, pela hipótese de indução e da equação $a_{ij}^{(k)} = \sum_{p=1}^{n} a_{ip}^{(k-1)} \cdot a_{pj}$, provamos que a afirmação é verdadeira para todo $k \geq 1$. ∎

Para que possamos ilustrar melhor o resultado do teorema acima, vamos obter a matriz de adjacência do problema dos amigos (Exemplo 2), e depois realizar as potências 2 e 3 da matriz de adjacência A desse mesmo problema.

Vimos como obter a matriz de adjacência A de um grafo pelo SageMath e vamos aprender como realizar as potências de uma matriz.

Declarado o grafo **G** do problema dos amigos (Exemplo 2), para obter a matriz de adjacência, acrescentamos os códigos $\mathbf{A} = \mathbf{G.adjacency\_matrix()}$, nomeando como **A** essa matriz; **show(A)**, para mostrar a matriz de adjacência **A**; e por fim, **show(A^k)**, para calcular e mostrar a potência k da matriz de adjacência **A**, para $k \geq 1$.

O código digitado no SageMath ficaria então assim:

Código 3.1 – Potência 2 e 3 da Matriz de adjacência no SageMath

```
G = Graph({'A': ['E','F'], 'B': ['C','D','E','G'], 'C': ['F','G'], 'D':
    ['G'], 'F': ['G']})

A = G.adjacency_matrix()   #Nomeando A como Matriz de Adjacência
G.vertices()   #Identificando a ordenação dos vértices
show(A)        #Mostrar a Matriz de Adjacência A
show(A^2)      #Calcular e Mostrar a Potência 2 de A
show(A^3)      #Calcular e Mostrar a Potência 3 de A
```

Ao executar o código acima, o SageMath apresenta os seguintes resultados:

```
['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 & 1 & 0 \end{pmatrix}$$

$$\begin{pmatrix} 2 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 4 & 1 & 1 & 0 & 2 & 2 \\ 1 & 1 & 3 & 2 & 1 & 1 & 2 \\ 0 & 1 & 2 & 2 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 2 & 1 & 1 \\ 0 & 2 & 1 & 1 & 1 & 3 & 1 \\ 1 & 2 & 2 & 1 & 1 & 1 & 4 \end{pmatrix}$$

$$\begin{pmatrix} 0 & 2 & 2 & 2 & 3 & 4 & 2 \\ 2 & 4 & 8 & 6 & 5 & 4 & 8 \\ 2 & 8 & 4 & 3 & 2 & 6 & 7 \\ 2 & 6 & 3 & 2 & 1 & 3 & 6 \\ 3 & 5 & 2 & 1 & 0 & 2 & 3 \\ 4 & 4 & 6 & 3 & 2 & 2 & 7 \\ 2 & 8 & 7 & 6 & 3 & 7 & 6 \end{pmatrix}$$

Observando as matrizes A^2 e A^3, vamos exemplificar o que representa cada entrada (i, j) dessas matrizes através do grafo desenhado.

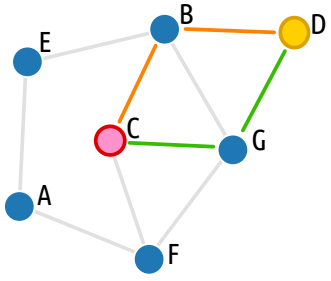
Matriz A^2

	A	B	C	D	E	F	G
A	2	1	1	0	0	0	1
B	1	4	1	1	0	2	2
C	1	1	3	2	1	1	2
D	0	1	2	2	1	1	1
E	0	0	1	1	2	1	1
F	0	2	1	1	1	3	1
G	1	2	2	1	1	1	4

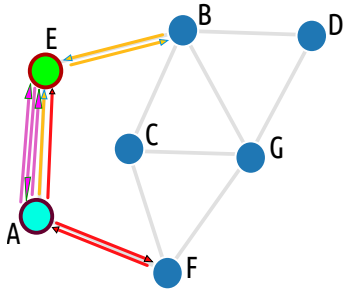
Matriz A^3

	A	B	C	D	E	F	G
A	0	2	2	2	3	4	2
B	2	4	8	6	5	4	8
C	2	8	4	3	2	6	7
D	2	6	3	2	1	3	6
E	3	5	2	1	0	2	3
F	4	4	6	3	2	2	7
G	2	8	7	6	3	7	6

Na Matriz A^2, a entrada $(4, 3)$ indica que existem 2 passeios de comprimento 2 ligando os vértices D e C, sendo eles $D - B - C$ e $D - G - C$.



Na Matriz A^3, a entrada $(1, 5)$ indica que existem 3 passeios de comprimento 3 ligando os vértices A e E, sendo eles $A - F - A - E$, $A - E - A - E$ e $A - E - B - E$.



Nessa mesma matriz, a entrada $(1, 4)$ indica que não existe passeio de comprimento 2 ligando os vértices A e D, já que para ir de A para D precisamos de um comprimento de, no mínimo, 4.

Nessa mesma matriz, a entrada $(6, 6)$ indica que existem 2 passeios de comprimento 3 ligando o vértice F a ele mesmo, a saber $F - C - G - F$ e $F - G - C - F$.

3.1 GRAFOS BIPARTIDOS

Um grafo é dito **bipartido**, se o conjunto dos vértices $V(G)$ pode ser particionado em dois subconjuntos X e Y em que cada aresta tem um extremo em X e outro extremo em Y. Nesse caso, a partição (X,Y) é chamada de *bipartição do grafo*.

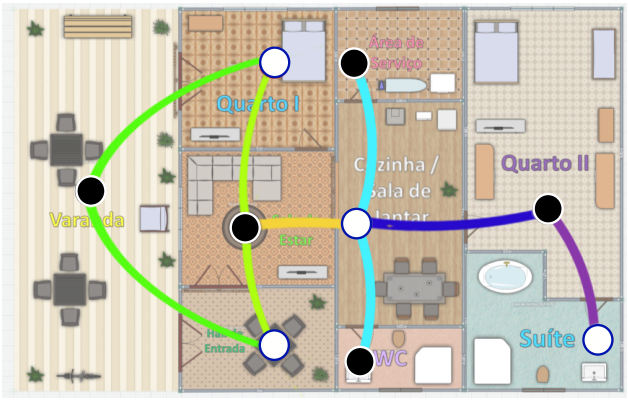
O exemplo a seguir introduz esse conceito de uma forma mais simples.

Exemplo 10 (Acessos aos Cômodos de uma Residência)**.** *Um arquiteto foi contratado para criar a planta baixa de uma casa com 2 quartos, um com acesso diretamente para a varanda e um outro quarto suíte maior que o primeiro, localizado nos fundos do terreno. Além disso, deveria ter um hall de entrada que teria acesso para a sala de estar e uma cozinha espaçosa, com acesso a uma área de serviço e a um banheiro social. De posse dessas exigências e sabendo que o terreno era retangular, o arquiteto propôs ao cliente a planta apresentada na figura a seguir:*

Figura 17 – Planta Baixa da Casa Solicitada pelo Cliente ao Arquiteto

O cliente pediu que o arquiteto destacasse na planta baixa todas as possibilidades de acesso aos cômodos que poderiam haver nesse projeto. Dessa forma, o arquiteto decidiu representar cada espaço da casa com um vértice e unindo esses vértices, caso houvesse acesso de um cômodo a outro. O arquiteto propôs, então, a seguinte figura:

Figura 18 – Grafo de Acesso aos Cômodos da Casa Solicitada pelo Cliente ao Arquiteto



Nesse exemplo, o conjunto dos vértices, representando os cômodos, poderia ser biparticionado, sendo X o conjunto dos cômodos {Varanda, Sala de Estar, Área de Serviço, WC, Quarto II} [vértices escuros na Figura 18] e o conjunto Y com os cômodos {Quarto I, Hall de Entrada, Cozinha/Sala de Jantar, Suíte} [vértices claros na Figura 18].

Embora a representação usando coloração de vértices pareça ter sido usada por acaso, vale ressaltar que a bipartição de um grafo é também conhecida como **bicoloração** de um grafo.

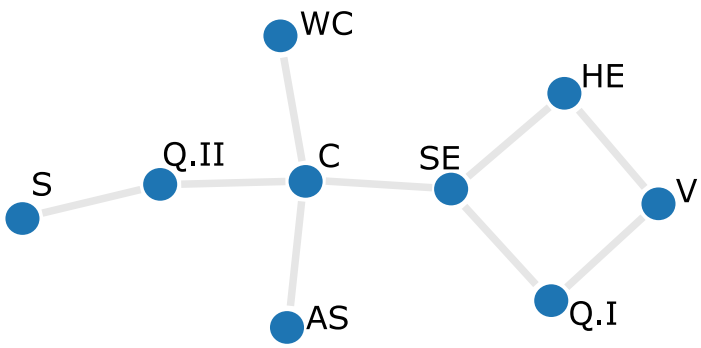
Programando no SageMath, reduzindo os nomes de cada cômodo a suas letras iniciais, escrevemos da seguinte forma:

Código 3.2 – Problema de Acesso aos Cômodos da Casa no SageMath

```
1 G = Graph ({'V': ['Q.I','HE'], 'Q.I': ['SE'], 'HE': ['SE'], 'SE': ['C'],
      'C': ['AS', 'WC', 'Q.II'], 'Q.II': ['S']})
2 show(G)
```

Executando o código anterior, obtemos uma forma de melhor visualizar os acessos existentes entre os cômodos da casa.

Figura 19 – Grafo Relativo ao Problema de Acesso aos Cômodos da Casa no SageMath



Para verificar se um grafo G é bipartido (em inglês, *is bipartite*) usando o SageMath, digitamos **G.is_bipartite()**, o que retorna *true* ou *false*.

No SageMath, é possível obter uma bipartição de um grafo bipartido G digitando **BipartiteGraph(G)**. Nomeando essa bipartição por **B**, por exemplo, ao digitar **B.left** e **B.right** conhecemos suas duas partições.

Usando o Código 3.2 do exemplo acima, já vimos que o grafo é bipartido e para obter suas partições, digitamos:

Código 3.3 – Biparticionando o grafo do Exemplo 10 no SageMath

```
1 G = Graph ({'V': ['Q.I','HE'], 'Q.I': ['SE'], 'HE': ['SE'], 'SE': ['C'],
      'C': ['AS', 'WC', 'Q.II'], 'Q.II': ['S']})
```

```
2
3  B = BipartiteGraph(G)
4  B.left
5  B.right
```

Ao executar, obtemos:

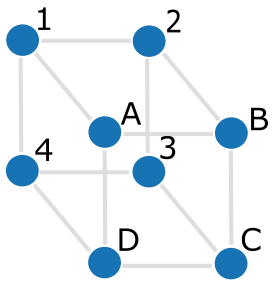
```
set(['C', 'HE', 'S', 'Q.I'])
set(['AS', 'Q.II', 'WC', 'SE', 'V'])
```

Na próxima seção, provaremos um teorema que nos dará subsídios para saber se um grafo é bipartido, e em caso positivo, como obter a bipartição desse grafo.

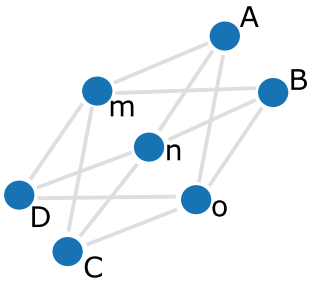
Um grafo bipartido completo é um grafo simples bipartido com bipartição (X,Y) em que cada vértice de X é ligado a cada vértice de Y. Se $|X| = m$ e $|Y| = n$, então denotamos esse grafo por $K_{m,n}$.

Figura 20 – Outros Exemplos de Grafos Bipartidos

(A) Grafo Bipartido - O Cubo



(B) Grafo Bipartido Completo - $K_{4,3}$



3.2 CICLOS

Vimos anteriormente que um passeio é fechado, se a origem e o fim desse passeio coincidem. Uma trilha fechada é um passeio sem repetição de arestas e cuja origem e fim desse passeio corresponde ao mesmo vértice. A uma trilha fechada no qual nenhum vértice, com exceção da origem e do fim, aparece mais de uma vez damos o nome de **ciclo**. Ou seja, para ser um ciclo precisa ser um caminho fechado com pelo menos uma aresta (com comprimento $k \geq 1$).

Um ciclo de comprimento 1 corresponde a um vértice com um laço. Um ciclo de comprimento 2 corresponde a duas arestas paralelas em que ambas têm um dos extremos no vértice de origem e o outro extremo no vértice interno desse caminho. Em um grafo

simples, o comprimento mínimo de um ciclo é 3, já que não pode haver laços nem arestas paralelas. Nesse último caso, chamamos de **ciclo simples**.

Um grafo ciclo é um grafo que consiste de um único ciclo, ou em outras palavras, todos os vértices desse grafo têm grau 2 e pertencem a um caminho fechado. Representamos um grafo ciclo com n vértices é chamado C_n.

Um grafo ciclo C_n (com comprimento n) é chamado de **n-ciclo**. Nomeamos um **n-ciclo** como **ciclo par**, se n for par (equivalente a dizer que o ciclo tem um número par de vértices); ou é nomeado como **ciclo ímpar**, se n for ímpar (equivalente a dizer que o ciclo tem um número ímpar de vértices).

Figura 21 – Exemplos de Ciclos e Grafos Ciclos

Na Figura 21, (A), o primeiro grafo é conexo e apresentam 2 ciclos, ambos de comprimento 3; e o segundo grafo é desconexo com um ciclo de comprimento 5. Já na parte (B), temos grafos ciclos de comprimento $k = 1$ [laço], $k = 2$ [arestas paralelas], $k = 3$ [um **3-ciclo** ou *triângulo*] e $k = 5$ [um **5-ciclo** ou *pentágono*], respectivamente.

Para criar um grafo ciclo (em inglês, *cycle graph*) com n vértices no SageMath, digitamos o seguinte código **G = graphs.CycleGraph(n)**.

Para saber se um grafo G é ciclo (em inglês, *cycle*), escrevemos **G.is_cycle()**, que retornará *true* ou *false*.

Um grafo que contém ao menos um ciclo é dito **grafo cíclico**, bem como um grafo que não contém ciclo de qualquer comprimento é chamado **grafo acíclico**.

Um grafo acíclico é chamado de floresta. Caso esse grafo acíclico seja conexo, chamamos de árvore. Dessa forma, uma floresta é uma coleção de árvores. Trataremos um pouco mais sobre árvores e florestas no Capítulo 4.

Todo grafo acíclico é um grafo bipartido, mas nem todo grafo cíclico é bipartido. O teorema a seguir generaliza esse fato.

Teorema 11 (Relação entre ciclos e grafos bipartidos)**.** *Um grafo G é bipartido se e somente se não contém ciclo ímpar.*

Demonstração: ($\Rightarrow$) Suponha que G é um grafo bipartido com bipartição (X, Y). O grafo G pode ter ou não um ciclo. Se ele não tem ciclo, então ele não tem um ciclo ímpar.

Caso ele tenha um ciclo, considere $C = v_0v_1 \dots v_kv_0$ esse ciclo de G. Sem perda de generalidade, assumiremos que $v_0 \in X$. Então, como G é bipartido e v_0v_1 é uma aresta desse ciclo, então $v_1 \in Y$. Da mesma forma, a aresta v_1v_2 indica que $v_2 \in X$. Generalizando, temos que $v_{2i} \in X$ e $v_{2i+1} \in Y$. Como $v_0 \in X$, então $v_k \in Y$. Segue $k = 2i+1$, para algum i, o que indica que C é um ciclo par.

($\Leftarrow$) Há duas possibilidades para o grafo G: ser conexo ou ser desconexo. Vamos analisar cada uma dessas possibilidades:

[CASO 1 : G é conexo] Consideremos x e y dois vértices de G e seja $D(x, y)$ o comprimento de um caminho mínimo ligando x e y.

Escolhemos arbitrariamente um vértice u do grafo G e definimos os conjuntos $X = \{v \in V(G) \mid D(u, v) \text{ é par}\}$ e $Y = \{v \in V(G) \mid D(u, v) \text{ é ímpar}\}$.

Queremos mostrar que quando G não contém ciclos ímpares, os conjuntos X e Y de vértices de G formam uma bipartição no grafo G que satisfaz a condição de que cada aresta em G possui um extremo em X e outro em Y e, dessa forma, o grafo G é bipartido.

Como G é conexo, pela definição todo vértice v está ligado a u, o que implica em $X \cup Y = V(G)$. O comprimento do caminho mínimo é único, logo não pode ser par e ímpar ao mesmo tempo. Conclui-se $X \cap Y = \emptyset$. Segue dessas duas conclusões que X e Y formam uma partição.

Suponha que $a, b \in X$. Vamos mostrar que $\{a, b\} \notin A(G)$. Sejam $C_a = (u, ..., a)$ e $C_b = (u, ..., b)$ os caminhos mínimos conectando o vértice u ao vértice a e conectando o vértice u ao vértice b, respectivamente.

Denotemos u_1 o último vértice comum a C_a e C_b. Como C_a e C_b são caminhos mínimos, um seção-(u, u_1) de ambos os caminhos são os caminhos mínimos de u a u_1 e, portanto, eles têm o mesmo comprimento. Se os comprimentos de C_a e C_b forem ambos pares, os comprimentos da seção-(u_1, a) C_a' de C_a e da seção-(u_1, b) C_b' de C_b devem ter a mesma paridade.

Seja I o ciclo formado pela concatenação $caminho - (a, b)\, C_b'^{-1}\, C_a'$. Esse ciclo I tem comprimento par. Se a fosse adjacente a b, então $C_a'^{-1}\, C_b' ba$ seria um ciclo de comprimento ímpar, o que contraria a hipótese. Dessa forma, concluímos que quaisquer dois vértices em X não são adjacentes. De maneira idêntica, prova-se que dois quaisquer vértices em Y não são adjacentes.

Portanto, toda aresta do grafo G possui um extremo no conjunto X e outro em Y, ou seja, G é bipartido.

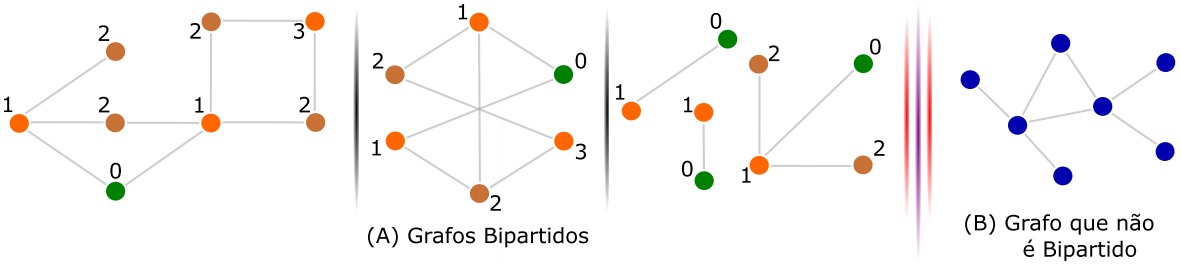
[CASO 2 : G é desconexo] Basta aplicar o raciocínio utilizado no caso 1 em cada uma das componentes conexas de G. ∎

Com a ajuda do Teorema 11 acima, temos uma forma de testar se um grafo é bipartido usando a ideia da paridade.

Se um grafo bipartido é conexo, a sua bipartição pode ser definida pela paridade das distâncias de qualquer vértice escolhido arbitrariamente v: um subconjunto consiste dos vértices a uma distância ímpar de v e o outro subconjunto consiste dos vértices a uma distância par de v.

Assim, uma forma de testar eficientemente se um grafo é bipartido, usando esta técnica de paridade de se atribuir vértices para os dois subconjuntos X e Y, separadamente a cada componente conectado do grafo e, em seguida, examinar cada aresta para verificar se ela tem extremos designadas para os diferentes subconjuntos.

Figura 22 – Testando se um Grafo é Bipartido



Os vértices em destaque representados por 0 foram escolhidos como referencial para calcular as distâncias em relação aos outros vértices e os números que aparecem em cada vértice equivale as distâncias em relação a esse vértice destacado.

Os dois primeiros grafos são conexos e o terceiro não é conexo. O quarto grafo não é bipartido, pois apresenta um ciclo ímpar e como vimos no Teorema 11, ele não pode ser biparticionado.

3.3 GRAFOS EULERIANOS E HAMILTONIANOS

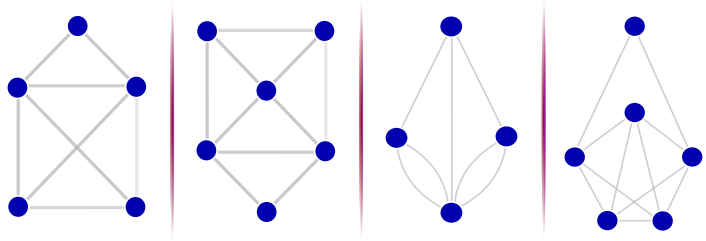
Chamamos **circuito euleriano** (ou **trilha de Euler** ou ainda **passeio de Euler**) a um passeio fechado que percorre todas as arestas de um grafo G exatamente uma vez.

Lembrando que a definição de passeio fechado é um passeio em que a origem e o fim coincidem. Um grafo é dito **euleriano** se possui um circuito euleriano.

Se tivermos um passeio aberto em um grafo G, ou seja, quando a origem e o fim não coincidem, tal que todas as arestas de G sejam percorridas exatamente uma vez, chamamos de **caminho euleriano** (em inglês, *eulerian path*).

Observando os grafos a seguir, é possível sair de um vértice e percorrer todas as arestas sem repetir nenhuma delas, retornando ao mesmo vértice inicial?

Figura 23 – É possível fazer um circuito euleriano por esses grafos?



De uma maneira mais prática, se pudermos desenhar esse grafo G sem retirar a ponta do lápis do papel, retornando ao vértice de início, então podemos fazer um circuito euleriano (em inglês, *eulerian circuit*).

O lema a seguir nos ajudará a entender o Teorema de Euler, que nos permitirá reconhecer as dificuldades que encontramos ao tentar fazer um circuito euleriano em alguns desses grafos.

Lema 12. *Se todo vértice de um grafo não necessariamente simples G tem grau maior ou igual a 2, então G contém um ciclo.*

Demonstração: Se G contém laços ou arestas paralelas, não há o que mostrar, pois naturalmente G contém um ciclo.

Focaremos apenas nos grafos simples. Partindo de um vértice u, percorremos uma trilha. Ao atingir um vértice qualquer, há duas hipóteses: ou se trata da primeira visita a esse vértice e podemos continuar a trilha, ou estamos chegando a um vértice já visitado, produzindo um ciclo. Como o número de vértices é finito, então o lema está provado. ■

Teorema 13. *(Teorema de Euler) Um grafo conexo, não necessariamente simples, G é euleriano se, e somente se, ele não contém vértices de grau ímpar.*

Demonstração:

($\Rightarrow$) Suponha que G tenha um circuito euleriano, então ele começa e termina no mesmo vértice u. Cada vez que esse circuito passa por um vértice interno v, duas arestas que incidem em v são usadas: uma para entrar e outra para sair.

Como o circuito euleriano contém todas as arestas de G, então o grau de v é par, para todo vértice $v \neq u$. Como o vértice u é o ponto de partida e de chegada desse circuito, então também tem grau par. Portanto, G não contém vértices de grau ímpar.

($\Leftarrow$) (*Indução em $|A(G)|$*) Quando $|A(G)| = 0$, ou seja, G não contém arestas, então G só possui um vértice e seu vértice tem grau par (zero) e naturalmente possui um circuito euleriano.

Suponhamos que o teorema seja válido para todos os grafos com menos que $|A(G)|$ arestas. Como G é um grafo conexo, todos os vértices têm grau maior que 2, já que os graus dos vértices são pares.

Pelo lema 12, G contém um ciclo, ou seja, uma trilha fechada. Dentre todas as trilhas fechadas do grafo G, escolhemos uma trilha T com comprimento máximo. Se essa trilha T tiver comprimento igual a $|A(G)|$, a prova desse teorema está finalizada.

Caso contrário, consideramos o grafo G' resultante da retirada das arestas de T. Como retiramos um número par de arestas de cada vértice de T, e que todos os vértices do grafo tem grau par (pela hipótese), pelo menos uma das componentes de G' tem um vértice em comum com T e tem todos os vértices com grau par.

Pela hipótese de indução, G' tem uma trilha fechada maior, concatenando T com a trilha em G', o que contraria a maximalidade na escolha de T. ∎

Se um grafo não é euleriano, mas tiver uma trilha aberta de comprimento $|A(G)|$, ou seja, um caminho euleriano, dizemos que ele é **semieuleriano**. O corolário a seguir apresenta um resultado interessante sobre grafos semieulerianos.

Corolário 14. *Um grafo conexo G, não necessariamente simples, é semieuleriano se, e somente se, tem no máximo dois vértices de grau ímpar.*

Demonstração:

($\Rightarrow$) Seja G um grafo semieuleriano, que começa em um vértice u e termina em um vértice v, com $v \neq u$, já que G é uma trilha aberta.

Logo tanto u quanto v possuem graus ímpares, pois a trilha aberta não termina por onde começou.

($\Leftarrow$) Seja o grafo G conexo com um par de vértices de grau ímpar, digamos que sejam os vértices u e v.

Pelo Teorema de Euler (13), acrescentando uma aresta ligando u a v, os graus de todos os vértices se tornam pares. Então existe uma trilha fechada de comprimento $|A(G)|+1$ com origem e fim em u (em outras palavras, trata-se de um grafo euleriano) e uma trilha aberta de comprimento $|A(G)|$ que começa em u e termina em v ao qual descreve um caminho semi-euleriano. ∎

Analisando os grafos apresentados na Figura 23 e de posse do que foi provado anteriormente, temos que:

1º Grafo: É facilmente desenhado sem tirar do papel a ponta do lápis. Se trata de um grafo semieuleriano, pois temos dois vértices de grau ímpar. Note que para dar certo precisamos começar por um vértice de grau ímpar e terminar no outro de grau ímpar.

2º Grafo: Aqui, mais uma vez, teríamos êxito ao desenhar usando a regra da ponta do lápis no papel. Se trata de um grafo semieuleriano, pois temos dois vértices de grau ímpar.

3º Grafo: Uma pessoa que tentasse desenhar conforme a regra, gastaria muito tempo e não teria êxito. Se trata de um grafo não euleriano. Observe que esse grafo apresenta os quatro vértices com grau ímpar. Esse 3º grafo foi essencial para o surgimento da Teoria dos Grafos.

Foi a partir de um famoso problema matemático que girava em torno das sete pontes de Königsberg (donde surge o grafo 3º, com os vértices representando ilhas e as arestas, as pontes ligando essas ilhas), onde se discutia nas ruas dessa cidade a possibilidade de atravessar todas as pontes sem repetir nenhuma.

Em 1736, o matemático Leonhard Euler se debruçou sobre o problema e provou que não existia caminho que possibilitasse tais restrições, donde se rendeu a homenagem no nome *trilha de Euler*.

4º Grafo: Como todos os vértices desse grafo têm graus pares, pelo teorema de Euler (13), temos que esse grafo é euleriano, ou seja, podemos desenhar conforme a regra.

De outra forma, um outro problema interessante gira em torno de buscar um caminho que percorra todos os vértices de um grafo G, ou seja, um passeio sem repetição de vértices e arestas, tendo passado por todos os vértices.

Esse caminho obedecendo a essa regra é dito um **caminho hamiltoniano** de G.

Se a origem e o fim desse caminho coincidem, damos o nome de **ciclo hamiltoniano**. Um grafo é dito **hamiltoniano** se contém um ciclo hamiltoniano. O nome é uma homenagem ao matemático William Hamilton, que estudou e divulgou esse problema.

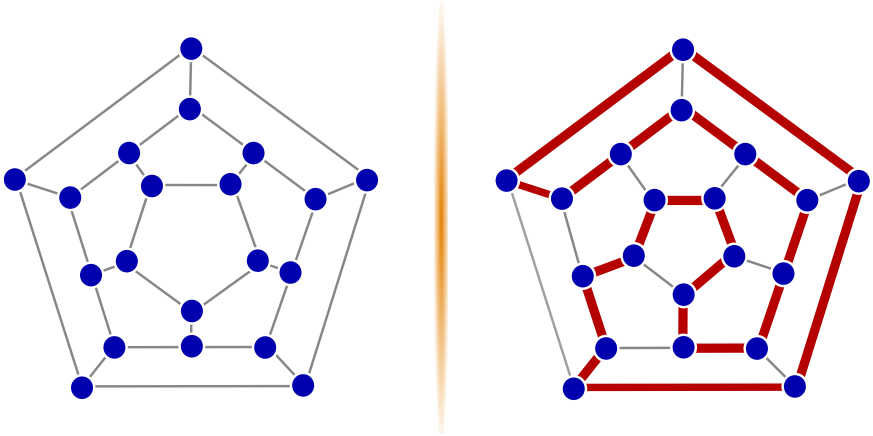
Conta-se que Hamilton inventou um jogo envolvendo um dodecaedro (sólido regular com 20 vértices, 30 arestas e 12 faces). Ele rotulou cada vértice do dodecaedro com o nome de uma cidade conhecida.

O objetivo do jogo era que o jogador viajasse "ao redor do mundo" determinando uma viagem circular que incluísse todas as cidades exatamente uma vez, com a restrição de que só fosse possível viajar de uma cidade a outra se existisse uma aresta entre os vértices correspondentes.

Observe que o grafo a seguir pode ser percorrido pela linha espessa atingindo todos os vértices sem repetir nenhum deles e retornando ao vértice de partida.

Se um grafo não contém um ciclo hamiltoniano, mas tiver um caminho entre dois vértices de forma que cada vértice do grafo seja visitado uma única vez, então este grafo é chamado **semi-hamiltoniano**.

Figura 24 – Um ciclo hamiltoniano do grafo do dodecaedro



Diferentemente dos grafos eulerianos, não se conhece nenhuma condição necessária e suficiente para garantir que um grafo seja hamiltoniano. Encontrar uma condição para esse problema é um dos principais problemas sem solução na Teoria dos Grafos.

O problema de reconhecer se um grafo é ou não hamiltoniano tem muitas aplicações em transporte, comunicação e planejamento, e por esse motivo constitui um dos mais estudados em Teoria dos Grafos.

É notável que todo grafo completo com mais de dois vértices é um grafo hamiltoniano. Isso porque, sendo $v_1, v_2, \ldots, v_k$ os vértices de G, sempre existe uma aresta entre qualquer par de vértices, e portanto, podemos percorrer, partindo de v_1, a sequência até v_k e retornar a v_1.

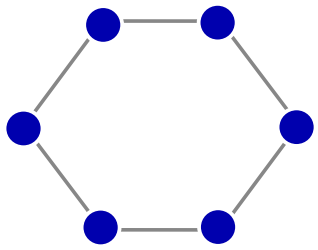
Como vimos, não temos teoremas que possam ser eficientes na afirmação de um grafo ser hamiltoniano ou não, porém dois teoremas nos ajudam nesse estudo. Aqui omitiremos suas demonstrações.

Teorema 15. *(Teorema de Dirac) Se G é um grafo simples com k vértices, onde $k \geq 3$, e $d(u) \geq \frac{k}{2}$, para cada vértice u, então G é hamiltoniano.*

Teorema 16. *(Teorema de Ore) Seja $G(V, A)$ um grafo simples com k vértices, onde $k \geq 3$, e $d(u) + d(v) \geq k$ para cada par de vértices não-adjacentes u e v, então G é hamiltoniano.*

Os dois teoremas acima são condições suficientes para um grafo ser hamiltoniano, mas não necessárias. Por exemplo, G pode ser simplesmente um ciclo, caso em que cada vértice tem exatamente grau 2, e ainda assim ser hamiltoniano. Na figura a seguir, o ciclo tem 6 vértices ($k = 6$) e não satisfaz ao Teorema 15, pois, para qualquer vértice u desse ciclo, $d(u) = 2 < \frac{k}{2} = \frac{6}{2} = 3$; nem satisfaz ao Teorema 16, pois a soma dos graus de quaisquer dois vértices não adjacentes nesse ciclo é igual a 4 que é menor que o número de vértices 6 desse ciclo.

Figura 25 – Um ciclo hamiltoniano que não satisfazem aos Teoremas 15 e 16.



No SageMath, podemos testar se um grafo G é euleriano (em inglês, *is eulerian*), através do código **G. is_eulerian()**, que retornará *true* ou *false*.

Retomando o Exemplo 2 sobre o problema dos amigos, digitamos **G. is_eulerian()** e obtemos o seguinte resultado:

```
False
```

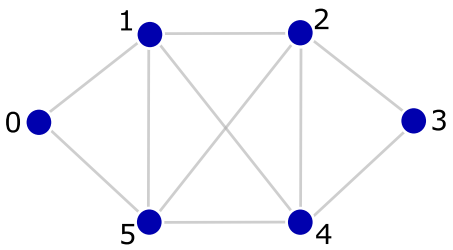
Se o grafo não for euleriano, como é o caso do grafo do problema dos amigos (Exemplo 2), podemos testar se ele é um grafo semieuleriano e para isso, escrevemos o código **G. is_eulerian(path=True)**, que retornará *false*, se ele não for semieuleriano; ou no caso em que o resultado seja *true*, o SageMath retorna um par de vértices (u, v) dizendo que existe um caminho euleriano com origem em u e fim em v.

Aplicando o código no problema dos amigos (Exemplo 2), temos como resultado o abaixo descrito:

```
('C', 'F')
```

Como o SageMath apresentou um par de vértices, então é sinal que G é semieuleriano e um caminho euleriano que inicia em C e termina em F está representado em $C - F - G - C - B - G - D - B - E - A - F$.

Considere o grafo G e o código que produziu a figura a seguir:



Código 3.4 – Código do grafo G

```
1 G = Graph({0: [1,5], 1: [2,4,5],
2     2: [3,4,5], 3: [4], 4: [5]})
3
4 show(G)
```

Se existir um circuito euleriano em um grafo G, podemos obter uma lista das arestas que formam esse circuito, através do código **G. eulerian_circuit()**, para listar as

arestas seguidas de seu rótulo (o rótulo padrão é *None*), ou **G.eulerian_circuit(labels=False)**, mostrando apenas a lista sem a rotulação das arestas.

No código do grafo G considerado, se digitarmos **G.eulerian_circuit()**, obtemos o resultado (1) abaixo; e se digitarmos **G.eulerian_circuit(labels=False)**, obtemos o resultado (2) abaixo.

(1)
```
[(0, 5, None), (5, 4, None), (4, 2, None), (2, 5, None),
(5, 1, None), (1, 4, None), (4, 3, None), (3, 2, None),
(2, 1, None), (1, 0, None)]
```

(2)
```
[(0, 5), (5, 4), (4, 2), (2, 5), (5, 1), (1, 4), (4, 3),
(3, 2), (2, 1), (1, 0)]
```

Nesse mesmo caso, podemos solicitar que, além de mostrar a lista de arestas desse circuito euleriano, fosse listado os vértices que são atingidos em sequência nesse circuito e para isso, escrevemos **A,V = G.eulerian_circuit(return_vertices=True);A;V**.

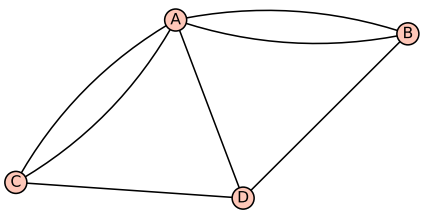
Nesse caso, estamos denominando a lista de arestas desse circuito por **A** e a lista de vértices desse circuito por **V** e por fim, pedimos para listar primeiro a lista de arestas e depois a lista de vértices.

Obtemos o seguinte resultado se usarmos os dois códigos anteriores [**A,V = G.eulerian_circuit(labels=False,return_vertices=True);A;V**] em relação ao grafo G considerado:

```
[(0, 5), (5, 4), (4, 2), (2, 5), (5, 1), (1, 4), (4, 3), (3,
2), (2, 1), (1, 0)]

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

Considere agora o grafo G abaixo sobre o problema das sete pontes de Königsberg e o código que o gera no SageMath apresentado a seguir:



Código 3.5 – Grafo das pontes de Königsberg

```
1 G = Graph({'A': ['B','B','C','C','D'],
      'D': ['B','C']})
2
3 plot(G)
```

Para saber se um grafo G é hamiltoniano (em inglês, *is hamiltonian*), entramos com o código **G.is_hamiltonian()**, que retornará *true* ou *false*.

No grafo G das pontes de Königsberg (Figura 3.3), digitando **G.is_hamiltonian()** retorna o seguinte resultado:

```
True
```

Já no problema das fronteiras do Nordeste Brasileiro (Exemplo 4), digitando o mesmo código, obtemos:

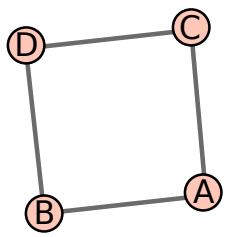
```
False
```

Podemos encontrar um ciclo hamiltoniano de um grafo G por meio de dois algoritmos: "tsp" ou "backtrack". Aqui não faremos comentários sobre o funcionamento desses algoritmos. Então, digitando o código **G.hamiltonian_cycle(algorithm='tsp')**, o SageMath retorna, se existir, uma mensagem dizendo que G possui ciclo hamiltoniano.

Se existir, o Sagemath dirá quantos vértices (aqui representado pelo X) existem nesse ciclo [mensagem retornada ⇒ **TSP from : Graph on X vertices**]. No exemplo do problema das pontes, temos que o código **G.hamiltonian_cycle(algorithm='tsp')** retorna a seguinte mensagem:

```
TSP from : Graph on 4 vertices
```

Digitando **show(G.hamiltonian_cycle(algorithm='tsp'))** é possível interagir com esse grafo, como mostra o resultado da aplicação desse código no problema das pontes de Königsberg.



Caso não exista um ciclo hamiltoniano, retornará um erro dizendo que o grafo dado não é hamiltoniano [mensagem retornada ⇒ **EmptySetError: The given graph is not Hamiltonian**]. Por exemplo, no grafo do problema das fronteiras do Nordeste Brasileiro,

vimos que esse grafo não é hamiltoniano e, portanto, digitando **G.hamiltonian_cycle()** ou **G.hamiltonian_cycle(algorithm='tsp')**, o SageMath retorna:

```
Error in lines XX-XX
Traceback (most recent call last):
   File ...
     raise EmptySetError("the given graph is not Hamiltonian")
EmptySetError: the given graph is not Hamiltonian
*** WARNING: Code contains non-ascii characters ***
```

Digitando o código **G.hamiltonian_cycle(algorithm='backtrack')**, o SageMath retorna um par (X, Y). Se X tem valor verdadeiro, então Y é um ciclo hamiltoniano. Se X tem valor falso, então Y é o caminho mais longo encontrado por esse algoritmo.

Aplicando o código acima no problema dos amigos (Exemplo 2), é retornado o resultado a seguir.

```
(True, ['D', 'G', 'C', 'F', 'A', 'E', 'B'])
```

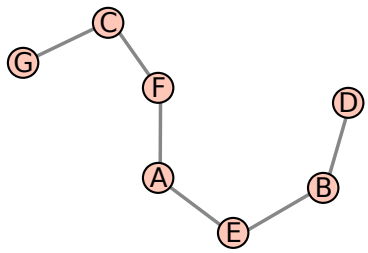
O primeiro termo indica que existe um ciclo hamiltoniano e um ciclo está representado no segundo termo. Observe que usar esse código, elimina a necessidade de usar o código **G.is_hamiltonian()**.

Uma outra possibilidade é encontrar um caminho hamiltoniano em um grafo G, desde que esse caminho exista. Para isso, usamos o código **G.hamiltonian_path()** e se existir, o SageMath retorna uma mensagem dizendo que o referido caminho hamiltoniano se trata de um subgrafo de G com o mesmo número de vértices k [mensagem retornada $\Rightarrow$ **Subgraph of (): Graph on** k **vertices**].

Aplicando no problema dos amigos (Exemplo 2), o SageMath retorna a seguinte mensagem:

```
Subgraph of (): Graph on 7 vertices
```

Se desejarmos visualizar e interagir com esse caminho hamiltoniano, basta digitarmos **show(G.hamiltonian_path())**. O resultado ao utilizar esse código no problema dos amigos (Exemplo 2) é mostrado a seguir.

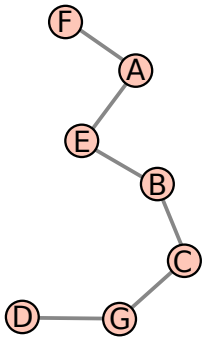


Caso G não seja semi-hamiltoniano, o SageMath não retorna resultado para esse código. Podemos também obter um caminho hamiltoniano entre dois vértices u e v de um grafo G, desde que exista. Para isso, usamos o código **G.hamiltonian_path(u, v)**, que retorna uma mensagem dizendo que o referido caminho hamiltoniano ligando os vértice u e v se trata de um subgrafo de G com o mesmo número de vértices k [mensagem retornada $\Rightarrow$ **Subgraph of (): Graph on** k **vertices**].

Aplicando no problema dos amigos (Exemplo 2), o SageMath retorna a seguinte mensagem:

```
Subgraph of (): Graph on 7 vertices
```

Se quisermos visualizar e interagir com esse grafo, por exemplo, buscando no problema dos amigos um caminho hamiltoniano partindo do vértice F e findando em D, digitamos **show(G.hamiltonian_path('F','D'))**, obtendo o caminho hamiltoniano abaixo.



3.4 O PROBLEMA DO MENOR CAMINHO

Um grafo é dito **valorado** ou **ponderado**, se as arestas desse grafo possuem "peso", ou melhor, as arestas possuem um valor atribuído. Esses valores podem representar

distâncias, custo de uma construção, custo de uma ligação, tempo gasto com um trajeto, dentre outros.

Dado um grafo valorado G, o problema do menor caminho consiste em encontrar um caminho entre vértices do grafo G com menor comprimento ou menor custo, sendo esse custo obtido pela soma dos pesos de cada aresta percorrida. O algoritmo que nos ajuda de forma eficiente a solucionar o problema foi criado pelo cientista da computação Edsger Wybe Dijkstra, em 1952.

O algoritmo de Dijkstra trabalha apenas com grafos valorados cujos valores sejam positivos e o objetivo é minimizar esses valores. Usaremos a nomenclatura P(v,w) ao peso da aresta que liga os vértices v e w e denotaremos (v,w) para representar o menor custo ou distância do vértice v ao vértice w. Definido um vértice de partida u, representaremos por $D(v)$ o valor de (u,v).

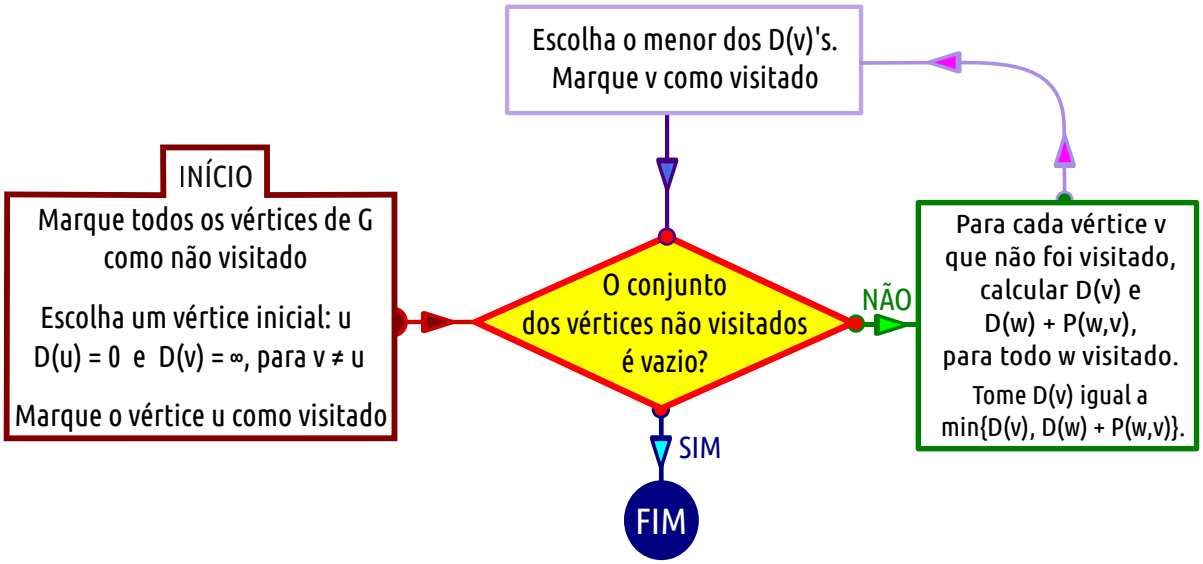
Algumas representações básicas devem ser destacadas no algoritmo de Dijkstra:

♢ Sempre que calcularmos o valor (distância, ou custo, ou tempo, etc) de um vértice a ele mesmo diremos que vale zero (D(u) = (u,u) = 0).

♢ Sempre que não atingirmos um determinado vértice por meio de uma aresta usaremos o símbolo de infinito (∞), ou seja, sempre que os vértices não forem adjacentes (D(v) = (u,v) = ∞, se v não é adjacente a u).

O fluxograma a seguir nos ajuda a entender os procedimentos para encontrar os caminhos mínimos de um vértice de um grafo valorado G a todos os outros vértices de G, usando o algoritmo de Dijkstra.

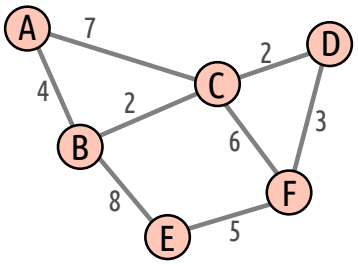
Figura 26 – Fluxograma do algoritmo genérico de Dijkstra



Para melhor compreender a utilização desse algoritmo, vamos considerar o exemplo a seguir:

Exemplo 17. *Considere o grafo valorado a seguir.*

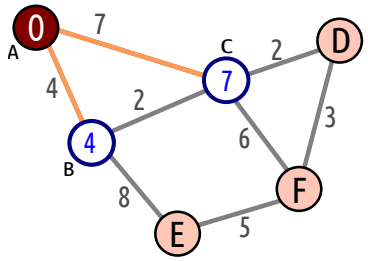
Figura 27 – Grafo valorado



Qual o menor caminho (com peso minímo) de um vértice escolhido aos outros vértices desse grafo?

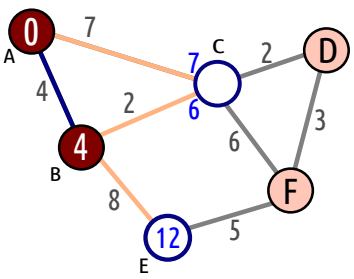
Escolhemos um vértice inicial, por exemplo, o vértice A, e marcamos ele como visitado. Calculamos a distância desse vértice a todos os vértices adjacentes (B e C), buscando o de menor distância, que é o vértice B com distância 4, marcando ele como visitado.

Figura 28 – Escolhendo o vértice inicial e calculando a distância aos vértices adjacentes (B e C).



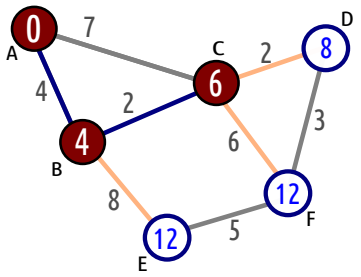
Agora calculamos as distâncias dos vértices visitados (A e B) aos seus vértices adjacentes (C e E). Buscando o de menor distância, encontramos o vértice C com distância 2 em relação a B, marcando ele como visitado.

Figura 29 – Calculando a distância aos vértices adjacentes (C e E) aos visitados (A e B).



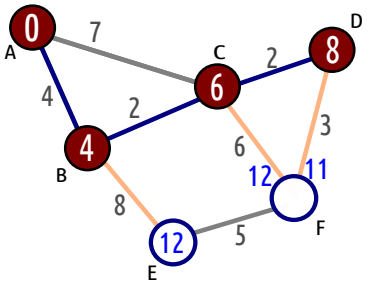
Continuamos calculando as distâncias dos vértices visitados (A, B e C) aos adjacentes a esses (D, E e F), buscando o de menor distância que, nesse caso, é o vértice D com distância 2 em relação ao vértice C e marcando ele como visitado.

Figura 30 – Calculando a distância aos vértices adjacentes (D, E e F) aos visitados (B e C).



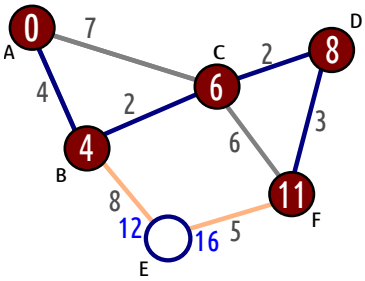
Como ainda temos vértices não visitados, continuamos nossa busca usando a mesma ideia anterior. Buscando dentre os vértices adjacentes aos já visitados, o que tem menor distância em relação aos vértices já visitados é o vértice F com distância 3 em relação ao vértice D. Marcamos ele como visitado.

Figura 31 – Calculando a distância aos vértices adjacentes (E e F) aos visitados (B, C e D).



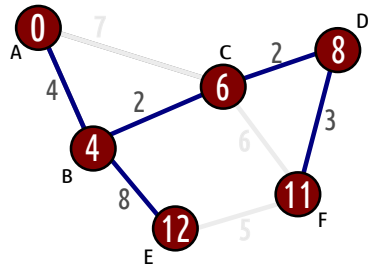
Como resta apenas o vértice E, precisamos calcular a distância em relação aos vértices B e F, sendo a menor distância 8 em relação ao vértice B, e marcamos ele como visitado.

Figura 32 – Calculando a distância ao vértice adjacente (E) aos visitados (B e F).



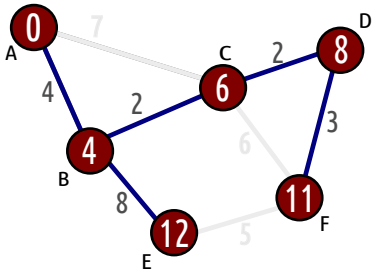
Por fim, obtemos o caminho de custo mínimo com os respectivos custos em relação ao vértice inicial.

Figura 33 – Menor caminho entre os vértices do grafo G.



Algumas observações importantes sobre o grafo final:

- O grafo final obtido pelo algoritmo de Dijkstra é uma árvore, ou seja, é um grafo conexo e sem ciclos, já que sempre que chegávamos em um vértice, nós excluíamos uma aresta, impedindo que ciclos fossem formados.
- O algoritmo de Dijkstra encontra o menor caminho de um vértice tomado como ponto de partida para todos os outros vértices. Ele **não** busca o menor caminho entre dois vértices quaisquer, mas é possível recomeçar o algoritmo com outro ponto de partida.



Por exemplo, o menor custo ligando os vértices de D e E é $3 + 5 = 8$, passando por F, e **não** $2 + 2 + 8 = 12$, passando por B e C, como sugere o grafo final.

Problemas de caminho mínimos costumam ser mais trabalhosos de solucionar se tivermos uma grande quantidade de vértices conectados. Com a ajuda do SageMath, é possível descobrir todos os menores caminhos partindo de um vértice inicial. Mas antes de aprendermos como usar essa ferramenta, precisamos aprender como declarar os pesos (em inglês, *weight*) ou valores de uma aresta.

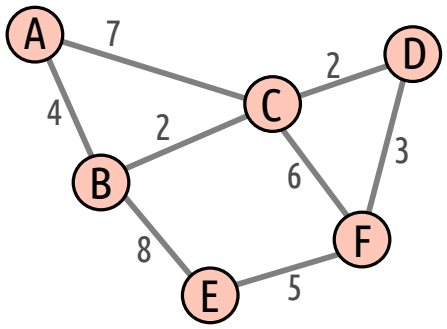
Primeiro, declaramos que o grafo G será valorado digitando **G = Graph (weighted = True)**.

No próximo passo, adicionamos arestas ao grafo G junto com seus respectivos pesos ou valores da seguinte forma: **G.add_edges([**$(u,\ v,\ p_1)$**, ⋯,** $(t,\ w,\ p_k)$**])**, onde cada

tripla indica que as duas primeiras entradas são vértices que se conectam, formando uma aresta de peso igual p_i inserido na terceira entrada da tripla.

Para obtermos os caminhos mínimos entre um vértice v escolhido como ponto de partida e todos os outros vértices desse grafo contendo pesos, digitamos o seguinte código: **G.shortest_paths(v, by_weight = True)** e o SageMath retornará uma lista informando os menores caminhos saindo de v até cada um dos vértices de G. Ao usar **by_weight = True**, o algoritmo retornado usado é o de Dijkstra.

No Exemplo 17, declaramos o grafo G e as arestas com seus respectivos pesos, conforme o grafo e o código a seguir:



Código 3.6 – Código do Problema do Menor Caminho

```
G = Graph (weighted = True)
G.add_edges([
    ('A','B',4),('A','C',7),('B','C',2),('B','E',8),
    ('C','D',2),('C','F',6),('D','F',3),('E','F',5)
  ])
G.shortest_paths('A', by_weight=True)
```

O SageMath apresenta o seguinte resultado:

```
{ 'A': ['A'], 'C': ['A', 'B', 'C'], 'B': ['A', 'B'], 'E': ['A',
'B', 'E'], 'D': ['A', 'B', 'C', 'D'], 'F': ['A', 'B', 'C', 'D',
'F'] }
```

Observe que o resultado apresentado pelo SageMath corresponde ao mesmo resultado obtido na nossa solução.

Podemos também obter o caminho mínimo entre dois vértices quaisquer u e v, digitando **G.shortest_path(u, v, by_weight = True)**.

Digitando **G.shortest_path('D', 'E', by_weight = True)** no SageMath, no código sobre o problema do menor caminho, é retornado como resultado o que segue.

```
['D', 'F', 'E']
```

Se não declararmos nos códigos apresentados a expressão **by_weight = True**, subentende-se que todas as arestas tem o mesmo peso e, portanto, encontrar um caminho mínimo equivale a encontrar o menor número de arestas ligando esses vértices.

Por exemplo, digitando **G.shortest_paths('A')**, no Exemplo 17, obtemos:

```
{ 'A': ['A'], 'C': ['A', 'C'], 'B': ['A', 'B'], 'E': ['A', 'B',
'E'], 'D': ['A', 'C', 'D'], 'F': ['A', 'C', 'F'] }
```

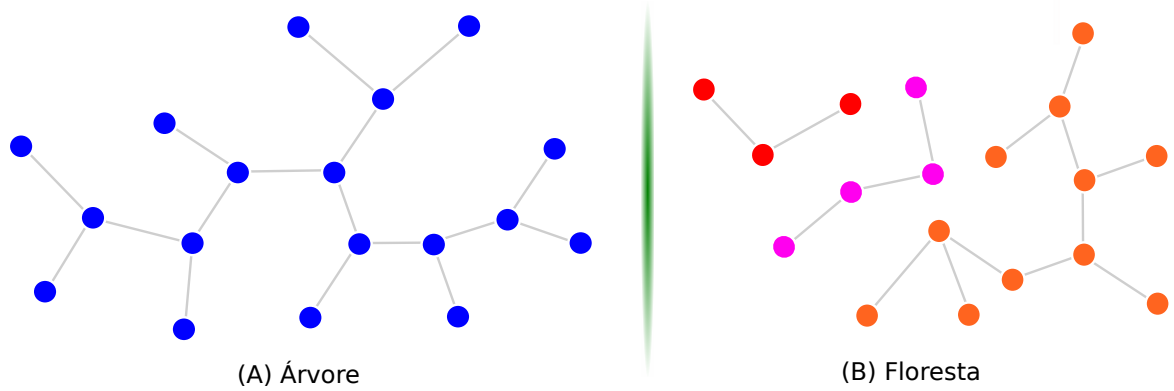
Para **G.shortest_path('B', 'F')**, devolve

```
['B', 'C', 'F']
```

4 ÁRVORES E FLORESTAS

Na Seção 3.2, definimos grafo acíclico como um grafo que não contém ciclos. Se G é um grafo acíclico, chamamos G de **floresta**. Uma **árvore** é um grafo acíclico conexo. Podemos, então, dizer que floresta é um grafo cujas componentes são árvores.

Figura 34 – Exemplo de Árvore e Floresta



Se escolhermos dois vértices quaisquer em uma árvore, só existe um único caminho ligando esses dois vértices, e isso é o que anuncia o nosso próximo Teorema.

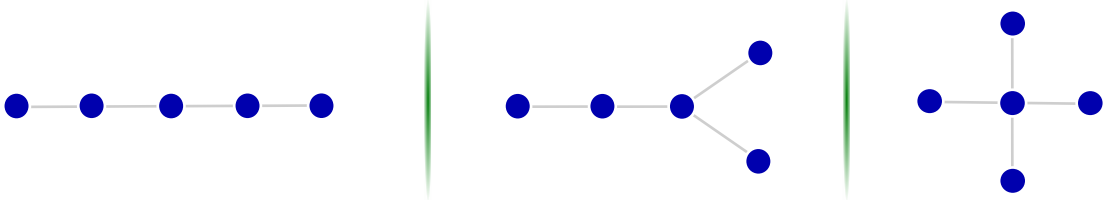
Teorema 18. *(Caminho Único em uma Árvore) Em uma árvore, quaisquer dois vértices são conectados por um único caminho.*

Prova por contradição: Seja G uma árvore e suponhamos que existam dois caminhos distintos Q_1 e Q_2 em G ligando os vértices u e v. Como $Q_1 \neq Q_2$, existe uma aresta a de Q_1 que conecta os vértices x e y, que não é aresta de Q_2.

É claro que o grafo $(Q_1 \cup Q_2) - \{a\}$ é conexo e contém um caminho-(x,y) Q. Mas $Q \cup \{a\}$ é um ciclo em grafo acíclico G, o que é um absurdo. ■

A figura a seguir apresenta todas as árvores com 5 vértices, a menos de isomorfismo.

Figura 35 – Árvores com 5 vértices



O próximo Teorema mostra a relação que existe entre o número de arestas e o de vértices em uma árvore. A fim de facilitar, escreveremos $|A|$ e $|V|$ em lugar de $|A(G)|$ e $|V(G)|$.

Teorema 19. *(Relação entre Vértices e Arestas em uma Árvore) Se G é uma árvore, então $|A(G)| = |V(G)| - 1$.*

Prova por Indução: *Indução em $|V|$.* Quando $|V| = 1$, G tem um único vértice e, portanto, $|A| = 0 = |V| - 1$.

Suponhamos agora que o Teorema seja válido para todas as árvores com menos de $|V|$ vértices, e seja G uma árvore com $|V| \geq 2$ vértices. Seja uv uma aresta de G ligando o vértice u ao vértice v.

$G - uv$ [árvore G excluída a aresta uv] não contém um caminho-(u,v), desde que uv é o único caminho-(u,v) em G.

Logo, $G - uv$ é desconexo e o número de componentes $\omega(G - uv) = 2$. Os componentes G_1 e G_2 de $G - uv$, por serem acíclicos, são árvores e além disso, cada um tem menos de $|V|$ vértices. Pela hipótese de indução, temos $|A(G_1)| = |V(G_1)| - 1$ e $|A(G_2)| = |V(G_2)| - 1$.

Temos $|A(G)| = |A(G_1)| + |A(G_2)| + 1 = |V(G_1)| - 1 + |V(G_2)| - 1 + 1 \Leftrightarrow |A(G)| = |V(G_1)| + |V(G_2)| - 1 \Leftrightarrow |A(G)| = |V(G)| - 1$. ∎

Uma árvore é dita **não trivial** se ela não possui vértices de grau 0, isto é, não é um vértice isolado. Uma consequência do Teorema 19 é o colorário a seguir:

Corolário 20. *Toda árvore não trivial tem, pelo menos, dois vértices de grau 1.*

Prova por contradição: Suponha por absurdo que o total de vértices de grau 1 seja 1. Seja G uma árvore não trivial. Pela definição, ela não possui vértices de grau 0, então temos $d(v) \geq 1$ para todo $v \in V$. Dessa forma, G possui pelo menos $(|V| - 1)$ vértices de grau maior que 1. Portanto, $\sum d(v) \geq 2(|V| - 1) + 1$.

Pelo Teorema 7, a soma dos graus de todos os vértices de um grafo G é igual ao dobro do número de arestas de G, e pelo Teorema 19, $|A| = |V| - 1$, se G é uma árvore, temos $\sum d(v) \geq 2(|V| - 1) + 1 \Leftrightarrow 2(|V| - 1) \geq 2|V| - 2 + 1 \Leftrightarrow 2|V| - 2 \geq 2|V| - 1 \Leftrightarrow -2 \geq -1$, o que é um absurdo.

Segue que o número mínimo de vértices v com grau 1 em uma árvore é 2. ∎

No SageMath, para saber um grafo G é uma árvore (em inglês, *tree*), digitamos **G.is_tree()** e retornará *true* ou *false*. Da mesma forma, para saber se G é uma floresta

(em inglês, *forest*), digitamos **G.is_forest()**.

Para gerar todas as árvores com um número fixo de vértices, não precisamos nomear vértices, nem a quem eles estavam conectados, basta digitarmos o seguinte código, substituindo no lugar de n pelo número de vértices da árvore:

Código 4.1 – Código para gerar todas as árvores com n vértices

```
G = graphs.trees(n)
for T in G: print(T.degree_sequence(), show(T))
```

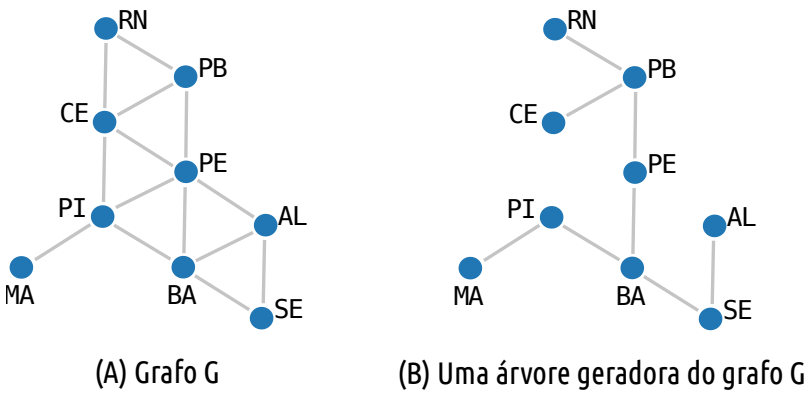
Nesse código **G** agora representa todas as árvores que podem ser geradas com n vértices. Para explorar cada árvore T em G, utilizamos o código **for T in G:** e solicitamos que fosse impresso a sequência de graus (em inglês, *degree sequence*) dos vértices dessa árvore T e que fosse mostrado cada uma dessas árvores: **print(T.degree_sequence(), show(T))**.

4.1 ÁRVORES GERADORAS

Definimos uma subárvore T de um grafo G como sendo qualquer subgrafo de G que seja uma árvore. Geralmente o termo *sub* é suprimido da palavra *subárvore* e dizemos apenas que T é uma árvore de G.

Uma árvore de um grafo G é **geradora** (em inglês, *spanning tree*) se contém todos os vértices de G. Um exemplo de uma árvore geradora pode ser observada na figura abaixo.

Figura 36 – Grafo G e uma árvore geradora desse grafo.



(A) Grafo G (B) Uma árvore geradora do grafo G

Pela definição de árvore, vimos que árvores são conexas, então todo grafo dotado de árvore geradora é conexo. Por outro lado, todo grafo conexo tem, pelo menos, uma árvore geradora. Toda árvore geradora de um grafo com $|V|$ vértices tem exatamente $|V| - 1$ arestas.

Para descobrir quantas árvores geradoras existem um grafo G, fazendo uso do SageMath, digitamos o código **G.spanning_trees_count()**, que retornará um número inteiro.

Há duas formas de encontrar uma árvore geradora de um grafo conexo: fazendo uma busca em profundidade ou fazendo uma busca em largura, que explicamos a seguir.

Um algoritmo de busca é um algoritmo que realiza a visita a todos os vértices de um grafo andando pelas arestas de um vértice a outro. Existem diversas formas de fazer essa tal busca. Cada algoritmo de busca é caracterizado pela ordem em que os vértices são visitados.

- **BUSCA EM PROFUNDIDADE**: Nesse tipo de busca, desejamos visitar todos os vértices e numerá-los na ordem em que são descobertos.

  O algoritmo genérico a seguir pode ser utilizado para realizar uma busca em profundidade. Para compreender melhor, precisamos indicar, para cada vértice, se ele foi visitado ou não.

  Seja um grafo G = (V,E) que contém n vértices. Seja também uma representação que indica, para cada vértice, se ele foi visitado ou não. Eis uma versão recursiva do algoritmo de busca em profundidade que visita todos os vértices:

  Dado um grafo G, siga os passos abaixo até que todos os vértices sejam visitados:

  (1) Marque todos os vértices de G como não visitados.

  (2) Escolha um vértice de G.

  (3) Marque esse vértice como visitado.

  (4) Se houver vértices não visitado adjacente a esse vértice, escolha um deles e volte ao passo (3). Caso contrário, volte ao último vértice visitado com vizinho não visitado e volte ao passo (3).

- **BUSCA EM LARGURA**: Nesse tipo de busca, usamos a ideia de distância $d(u, v)$ entre vértices u e v que é o comprimento do menor caminho entre u e v.

  Começamos por um vértice escolhido, denominado **raiz**, exploramos todos os vértices vizinhos, ou seja, que estão a uma mesma distância da raiz.

  Então, para cada um desses vértices mais próximos, exploramos os seus vértices vizinhos inexplorados e assim por diante, até que ele encontre o alvo da busca, atingindo todos os vértices de G.

Pelo SageMath, é possível encontrar todas as árvores de um grafo G dado, digitando o código **show(G.spanning_trees())** e será listado uma a uma as árvores desse grafo.

Se digitarmos **G.random_spanning_tree()**, o SageMath escolherá aleatoriamente uma dessas árvores e mostrará uma lista com as arestas que pertencem a essa árvore.

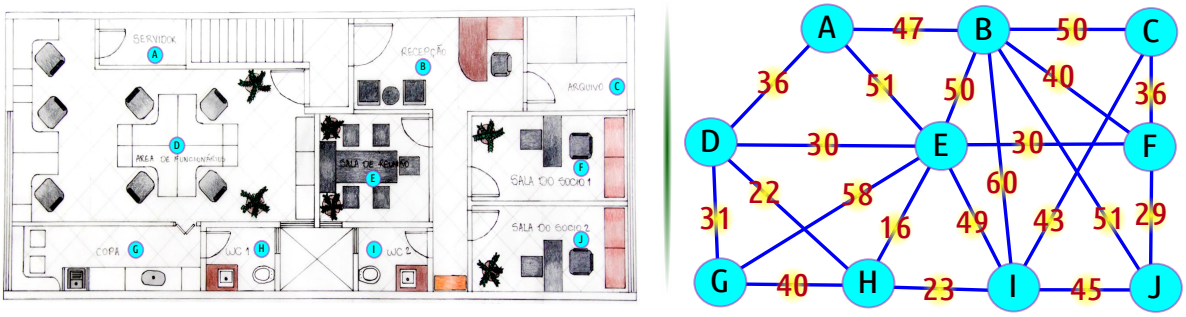
4.2 O PROBLEMA DE CONEXÃO DE PESO MÍNIMO

Considere o seguinte problema:

Exemplo 21. *(Instalação da Tubulação de Condicionador de Ar) Com a chegada do verão, o calor vem incomodando os funcionários de uma empresa de contabilidade. Essa empresa decide instalar uma central de condicionador de ar para melhor acomodação dos seus funcionários e pretende distribuir para todos os cômodos desse prédio o ar condicionado produzido pela central. Alguns fatores são levados em conta na hora de instalar as tubulações, entre eles a distância entre os pontos de distribuição, a necessidade de cada setor dessa empresa e os custos da obra para a passagem das tubulações, por exemplo. Esses fatores foram contabilizados por meio de valores sendo que quanto maior esse valor, maior é o custo de instalação dessas tubulações.*

A figura abaixo apresenta a planta baixa e o grafo relacionado a esse problema, com seus respectivos valores (ou pesos).

Figura 37 – Planta baixa do escritório de contabilidade e grafo valorado do problema



Deseja-se atingir todas os pontos com o mínimo de custo com a construção dessas tubulações. Dessa forma, que tubulações devemos construir para alcançar esse objetivo?

Nesse Exemplo, buscamos nesse grafo conexo uma solução onde todos os vértices se conectem com custo mínimo e, portanto, com o menor número de arestas, o que implica no conceito de árvore geradora. Dessa forma, se G é um grafo com k vértices, essa árvore terá $k-1$ arestas, já que atinge todos os vértices.

Já vimos como encontrar uma árvore geradora de um grafo G, mas a questão é: será que ela é mínima (com custo mínimo)?

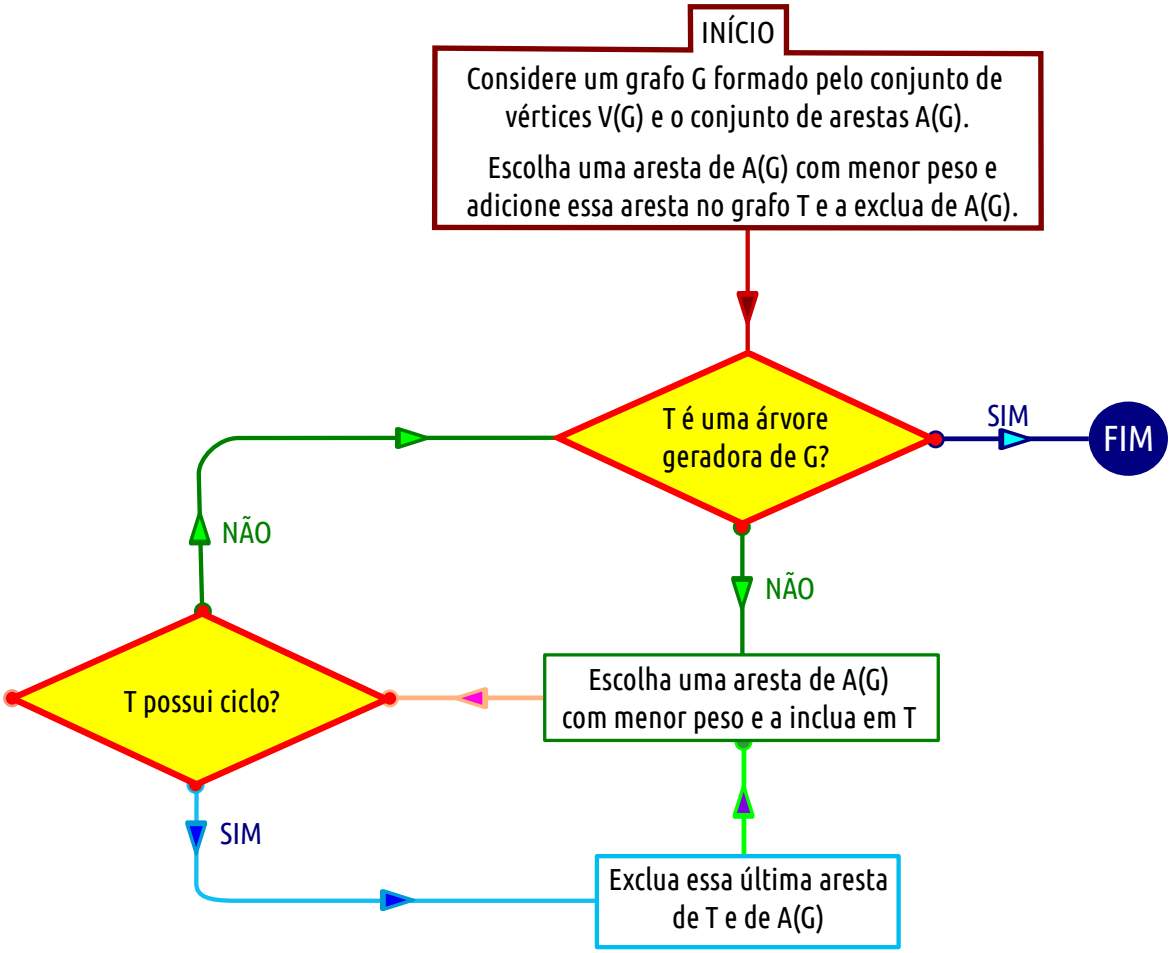
Então, a solução para o Exemplo 21 acima é uma árvore geradora e precisamos encontrar a árvore que tenha o menor custo total, onde esse custo é dado pela soma dos custos das arestas escolhidas (mínima). Dizemos, então, que temos uma solução ótima.

Para isso, conheceremos dois algoritmos que podem nos auxiliar nesse processo: o algoritmo de Kruskal e o algoritmo de PRIM.

◊ **Algoritmo de Kruskal:** Consiste em escolher as arestas com menores custos, uma a uma, com o cuidado de não formar ciclos. Dessa forma, obtemos uma árvore geradora mínima para qualquer grafo.

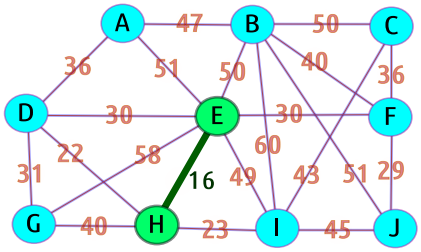
Esse algoritmo é apresentado no fluxograma a seguir:

Figura 38 – Fluxograma do algoritmo de Kruskal

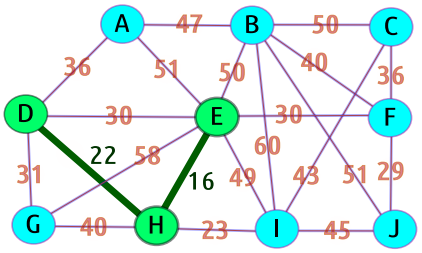


No Exemplo 21 acima, usando o algoritmo de Kruskal, temos a seguinte solução:

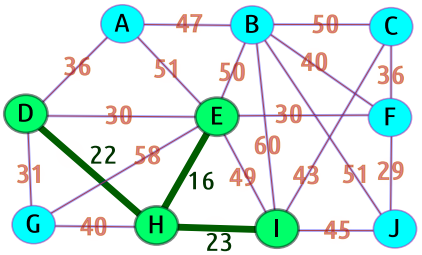
Primeiramente, escolhemos a aresta com menor custo: a que liga o vértice E ao H, com custo 16. Como essa é nossa primeira aresta, ela não forma ciclo e acrescentamos ela a nossa árvore T.



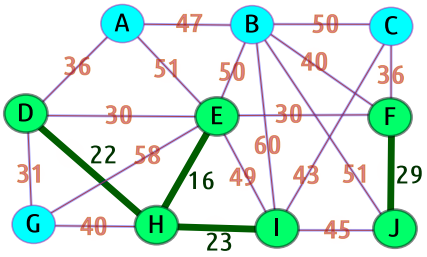
Buscamos agora a próxima aresta de menor custo que não esteja em T: a que liga H a D, com custo 22. Como ela não forma um ciclo com a aresta de T, então a incluímos em T.



A próxima aresta de menor custo que não esteja em T é a que liga H a I, com custo 23. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

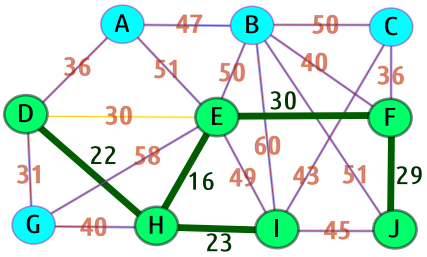


Agora, procuramos outra aresta de menor custo que não esteja em T: a que liga F a J, com custo 29. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

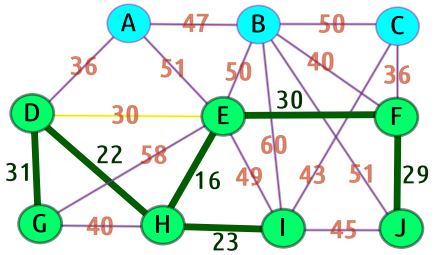


Seguindo o algoritmo, procuramos outra aresta de menor custo que não esteja em T, mas há um empate entre as arestas que ligam D a E e E a F, ambos com custo 30. Podemos escolher uma delas. Optando pela aresta que liga D a E, obtemos um ciclo (D - H - E - D) com as arestas de T. Nesse caso, nós a descartamos.

Usamos, portanto, a aresta que liga os vértices E e F. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

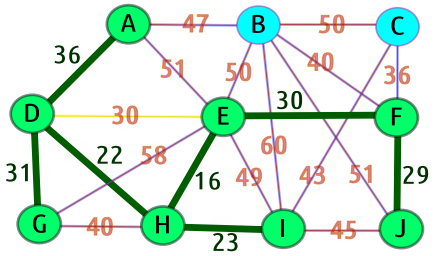


A aresta de menor custo que não está em T é a aresta que liga D a G, com custo 31. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

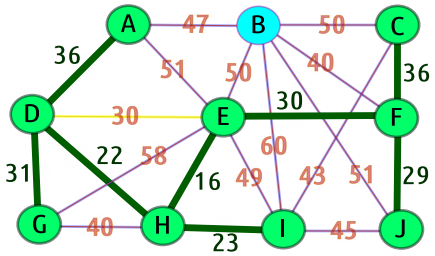


Seguindo o algoritmo, procuramos outra aresta de menor custo que não esteja em T, mas há um empate entre as arestas que ligam A a D e C a F, ambos com custo 36. Podemos escolher qualquer uma delas.

Suponha que optamos pela aresta que liga A a D. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

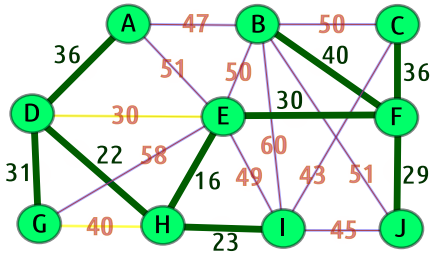


Agora optamos pela outra aresta que liga C a F, com 36 de custo. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.

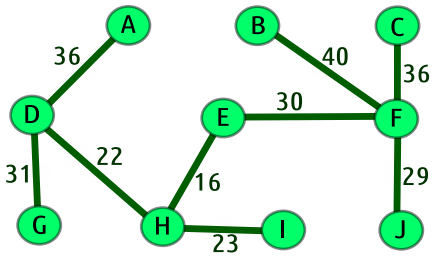


Continuando o algoritmo, procuramos outra aresta de menor custo que não esteja em T, mas há outro empate entre as arestas que ligam B a F e G a H, ambos com custo 40. Esta segunda aresta formaria um ciclo D - G - H - D, e, portanto, a descartamos.

A aresta que liga B a F, com 36 de custo, é escolhida. Como ela não forma um ciclo com as arestas de T, então a incluímos em T.



Finalizamos a nossa solução e encontramos a árvore geradora mínima, já que atingimos todos os 10 vértices do problema e as $10 - 1 = 9$ arestas que compõem essa árvore.



O custo total é dado por:

$16+22+23+29+30+31+36+36+40 = 263.$

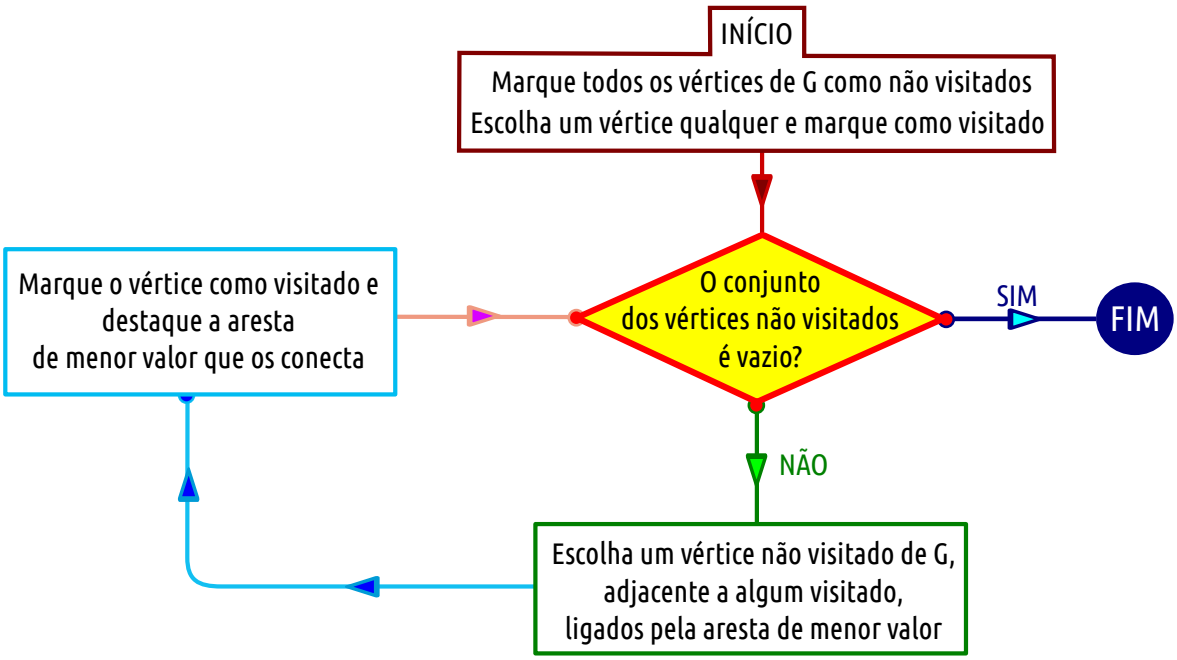
◊ **Algoritmo de Prim:** Funciona semelhante ao algoritmo de Kruskal.

No algoritmo de Kruskal, iniciamos com diversas árvores separadas, a saber, no começo, cada vértice é uma árvore, e vamos conectando essas árvores através das arestas de menor custo até formar uma árvore geradora.

No algoritmo de Prim, iniciamos com uma árvore formada por um único vértice (um qualquer do grafo) e vamos adicionando à essa árvore, a cada passo, o vértice mais próximo dela cuja aresta possui menor custo.

Esse algoritmo é apresentado no fluxograma a seguir:

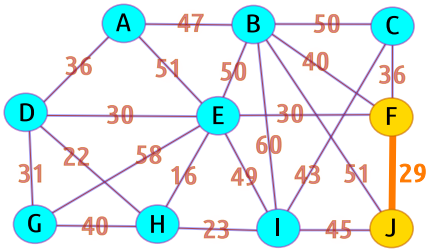
Figura 39 – Fluxograma do algoritmo de Prim



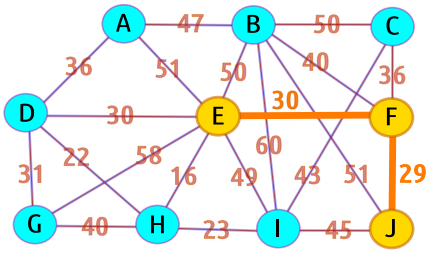
No Exemplo 21 acima, usando o algoritmo de Prim, temos a seguinte solução:

Primeiramente, escolhemos um vértice qualquer, por exemplo, J, e adicionamos à árvore T.

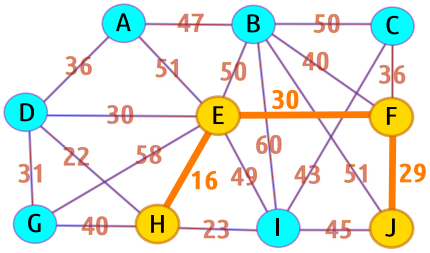
Buscamos o vértice mais próximo de T ligado a um vértice de T pela aresta de menor valor (custo). Nesse caso, o vértice é F e a aresta que liga o vértice J a F tem custo 29. Adicionamos esse vértice F à árvore T e a aresta que os conecta.



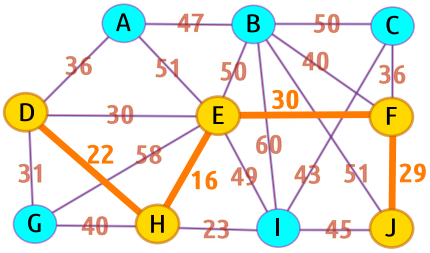
Buscamos agora o próximo vértice ligado a um vértice de T com menor custo: vértice E conectado a F, com custo 30. Adicionamos E e a aresta que conecta E a F na árvore T.



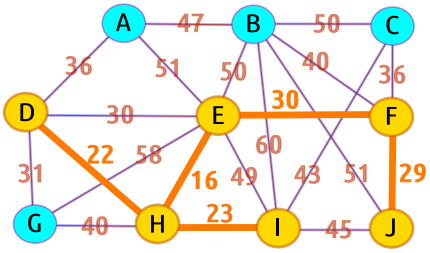
Procuramos outro vértice mais próximo de T ligado por aresta de menor custo. Nesse caso temos o vértice H e a aresta tem custo 16. Adicionamos H e a aresta que conecta E a H na árvore T.



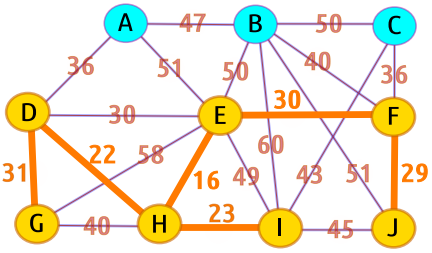
Agora, procuramos um vértice ligado a um vértice de T, cuja aresta que os conecta tem menor custo. Nesse caso, temos o vértice D e a aresta que liga D a E, com custo 22. Incluímos D e a aresta que conecta D a E na árvore T.



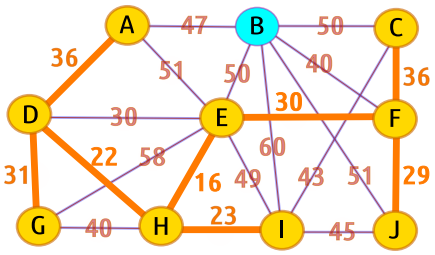
O próximo vértice atingido, conectado a um vértice de T, cuja aresta tem custo mínimo, é I, onde a aresta tem custo 23. Incluímos em T o vértice I e a aresta que os conecta.



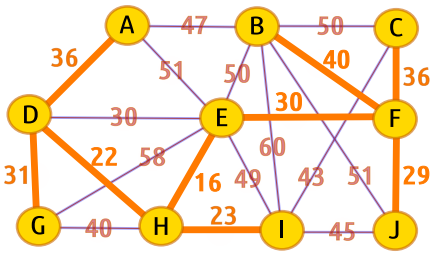
O vértice atingido por um vértice de T, cuja aresta tem custo mínimo, é G, onde a aresta tem custo 31. Incluímos em T o vértice G e a aresta que os conecta.



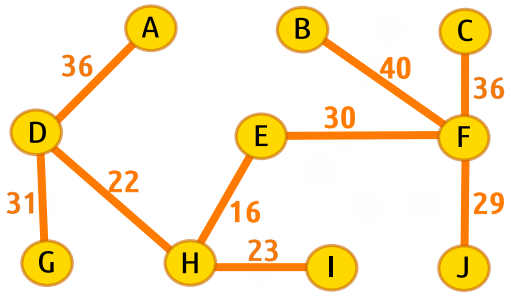
Buscamos o vértice, mais próximo de T, ligado por uma aresta de custo mínimo, e nesse caso, temos A e C, ambos com aresta cujo custo é 36. Incluímos em T os vértices A e C e a aresta que os conecta a T.



Procuramos o vértice mais próximo de T ligado por uma aresta de custo mínimo. Nesse caso, temos um último vértice B, cuja aresta tem custo mínimo 40. Incluímos em T o vértice B e a aresta de custo mínimo que os conecta a T.



Finalizamos a nossa solução e encontramos a árvore geradora mínima, já que atingimos todos os 10 vértices do problema.



Os dois algoritmos solucionam facilmente problemas de conexão de peso mínimo, até mesmo em grafos grandes. O que os diferenciam é o que escolhemos para incluir na árvore T para torná-la geradora mínima: no algoritmo de Kruskal, escolhemos a aresta com menor custo que não forma ciclo e no algoritmo de Prim, escolhemos o vértice que está ligado a árvore T pela aresta de menor custo.

Essa ideia de, a cada passo, escolher a opção que pareça a melhor possível, sem se preocupar com o que irá acontecer nos próximos passos devido as possíveis consequências dessa escolha, faz com que esses algoritmos sejam conhecidos como **algoritmos gulosos** (ou míopes).

No SageMath, é possível encontrar as arestas de uma árvore geradora mínima (em inglês, *minimal spanning tree*). Digitando **G.min_spanning_tree()**, obtemos as arestas que compõem a árvore geradora de custo mínimo, se existir. Caso contrário, é retornado uma lista vazia.

Já vimos como declarar um grafo valorado na Seção 3.4 sobre o problema do caminho mínimo e agora podemos computar a árvore geradora mínima usando um dos dois algoritmos apresentados anteriormente:

- declarado o grafo G valorado, usando o código **G.min_spanning_tree()** ou com **G.min_spanning_tree(algorithm = "Prim_Boost")**, o algoritmo utilizado para chegar à solução será o algoritmo de Prim; e
- se entrarmos com **G.min_spanning_tree(algorithm = "Kruskal")**, o algoritmo utilizado para chegar à solução será o algoritmo de Kruskal.

Vamos aplicar os códigos acima no problema da instalação da tubulação de condicionador de ar (Exemplo 21). Primeiro, precisamos declarar que se trata de um grafo com pesos. O código a seguir declara esse grafo valorado e adiciona as arestas com seus respectivos pesos. Além disso, solicita a árvore geradora mínima pelos dois algoritmos.

Código 4.2 – Código do grafo de Instalação das Tubulações dos condicionadores de ar

```
G = Graph (weighted = True)
G.add_edges([('A','B',47),('A','E',51),('A','D',36),
('B','E',50),('B','I',60),('B','J',51),('B','F',40),('B','C',50),
('C','F',36),('C','I',43),
('D','E',30),('D','G',31),('D','H',22),
('E','F',30),('E','G',58),('E','H',16),('E','I',49),
('F','J',29),('G','H',40),('H','I',23),('I','J',45)])

G.min_spanning_tree(algorithm='Prim_Boost')
G.min_spanning_tree(algorithm='Kruskal')
```

O SageMath apresenta o seguinte resultado:

```
[('A','D', 36), ('B','F', 40), ('C','F', 36), ('D','G', 31),
('D','H', 22), ('E','F', 30), ('E','H', 16), ('F','J', 29),
('H','I', 23)]
[('A','D', 36), ('B','F', 40), ('C','F', 36), ('D','G', 31),
('D','H', 22), ('E','F', 30), ('E','H', 16), ('F','J', 29),
('H','I', 23)]
```

Observe que os resultados apresentados pelos dois algoritmos são semelhantes e correspondem aos mesmos resultados obtidos na nossa resolução.

Suponha agora que desejamos obter uma árvore geradora cujo custo total seja máximo, a qual chamamos **árvore geradora máxima**. Vamos implementar uma função que retorna a árvore geradora de custo máximo e esse custo.

Esse problema se trata de uma adaptação do problema clássico de achar a árvore geradora mínima e para solucioná-lo, usamos uma versão adaptada do algoritmo de Kruskal. Primeiramente ordenaremos as arestas em ordem decrescente de peso (custo), depois vamos acrescentando as arestas com custo maior de forma a evitar ciclos até obter uma árvore geradora, que por sua vez terá custo máximo.

Dividiremos nossa função em duas partes: a primeira listará os pesos, contará o custo total das arestas e organizará as arestas em ordem crescente de peso e a segunda parte contará o custo máximo e montará a árvore geradora de custo máximo, eliminando as arestas de custo mínimo, desde que não torne o grafo desconexo, até obter uma árvore.

Inicialmente declaramos um grafo valorado sem arestas (**A = Graph(weighted= True)**), armazenamos o número de arestas do grafo de entrada X em uma variável n (**n = len(X.edges())**), declaramos duas listas vazias (**L = []; arestas_ord = []**) que usaremos para armazenar os pesos e as arestas ordenadas pelos pesos na ordem decrescente, respectivamente.

Percorremos as arestas do grafo X (**for k in X.edges():**), armazenamos os custos de cada aresta na lista L (**L.append(k[2])**).

Agora, ordenaremos, nessa etapa, as arestas pelos seus respectivos pesos em ordem decrescente, procuramos o maior valor na lista L e para isso usamos a função **max** do Python (**m = max(L)**).

Percorremos as arestas do grafo X (**for i in X.edges():** e procuramos as arestas cujo peso coincide com o maior valor da lista L (**if i[2] == m:**), adicionamos essas arestas na lista arestas_ord (**arestas_ord.append(i)**), retiramos seu custo da lista L (**L.remove(m)**) e descontamos 1 unidade no número de arestas (**n = n - 1**).

Repetimos essa etapa até que zere o número de arestas, ou seja, repete a etapa enquanto n for diferente de zero (**while n != 0:**).

O resultado final dessa etapa é a lista **arestas_ord** com as arestas ordenadas pelo custo na ordem decrescente.

Na segunda parte, consideramos inicialmente o custo máximo igual a zero (**custo_max = 0**). Percorremos as arestas da lista arestas_ord (**for k in arestas_ord:**) e a cada aresta, adicionamos tal aresta ao grafo A (**A.add_edge(k)**) e adicionamos seu peso ao custo máximo (**custo_max = custo_max + k[2]**).

Precisamos saber se o grafo A contém ciclos, e para tal, consultamos se o grafo A é uma floresta (**if A.is_forest():**, já que florestas não contém ciclos.

Em caso positivo, verificamos se o grafo A é uma árvore geradora, ou seja, se o número de arestas de A é igual ao número de vértices de X menos 1 (**if len(A.edges()) == len(X)-1 and A.is_tree()**), imprimimos o custo máximo dessa árvore geradora máxima (**print "Custo Máximo:", custo_max**), visualizamos o grafo A correspondente a essa árvore (**show(A)**) e paramos de percorrer as outras arestas da lista arestas_ord (**break**); caso negativo, excluímos a aresta analisada (**A.delete_edge(k)** e diminuímos seu custo do custo máximo (**custo_max = custo_max - k[2]**).

Código 4.3 – Função que retorna a árvore geradora máxima e seu custo

```
def arv_gerad_max(X):
    A = Graph(weighted=True)
    n = len(X.edges())
    L=[]
    arestas_ord=[]
    for k in X.edges(): L.append(k[2])

    while n != 0:
        m = max(L)
        for i in X.edges():
            if i[2] == m:
                arestas_ord.append(i); L.remove(m); n = n - 1

    custo_max = 0
    for k in arestas_ord:
        A.add_edge(k)
        custo_max = custo_max + k[2]

        if A.is_forest():
            if len(A.edges()) == len(X)-1 and A.is_tree():
                print "Custo Máximo:", custo_max; show(A); break
        else:
            A.delete_edge(k); custo_max = custo_max - k[2]

```

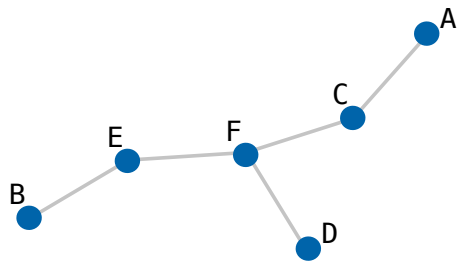
Retomando o grafo do Exemplo 17 sobre o problema do menor caminho, declaramos seus respectivos grafos e aplicamos a função **arv_gerad_max()** conforme códigos a seguir.

```
G = Graph ([('A','B',4), ('A','C',7), ('B','C',2), ('B','E',8),
  ('C','D',2),('C','F',6),('D','F',3),('E','F',5)], weighted = True)])

arv_gerad_max(G)
```

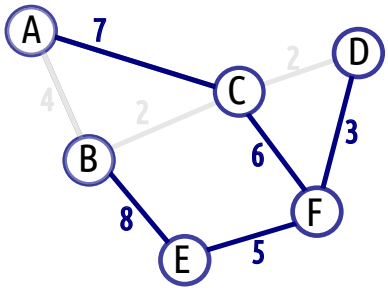
O SageMath apresenta o seguinte resultado:

```
Custo Máximo: 29
```



Melhor visualizando, temos:

Figura 40 – Árvore geradora máxima do grafo do problema do menor caminho



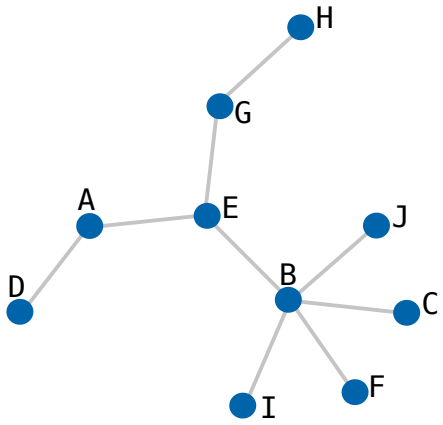
Retomando o grafo do Exemplo 21 sobre o problema de instalação do condicionador de ar, declaramos seu grafo e aplicamos a função **arv_gerad_max()** conforme código a seguir.

```
M = Graph (weighted = True)
M.add_edges([('A','B',47),('A','E',51),('A','D',36),
('B','E',50),('B','I',60),('B','J',51),('B','F',40),('B','C',50),
('C','F',36),('C','I',43),
('D','E',30),('D','G',31),('D','H',22),
('E','F',30),('E','G',58),('E','H',16),('E','I',49),
('F','J',29),('G','H',40),('H','I',23),('I','J',45)])

arv_gerad_max(M)
```

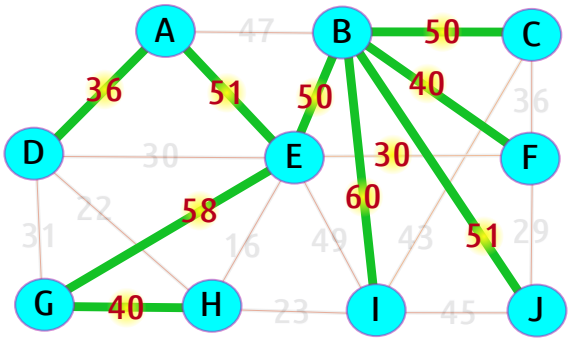
O SageMath apresenta o seguinte resultado:

```
Custo Máximo: 436
```



Melhor visualizando, temos:

Figura 41 – Árvore geradora máxima do problema de instalação do condicionador de ar



5 PROPOSTA PEDAGÓGICA

Essa proposta pedagógica visa abordar os conteúdos que foram tratados aqui nesse trabalho de modo que o aluno possa aliar os conhecimentos matemáticos adquiridos envolvendo Teoria dos Grafos ao poder da lógica e da ciência da computação, entendendo a programação como aliado na resolução de problemas.

O foco aqui são alunos do Ensino Médio, de preferência os da segunda série do Ensino Médio, após terem estudado Matrizes e Análise Combinatória, conteúdos programáticos desse nível, porém o ensino de lógica de programação fazendo uso de uma linguagem de programação poderá ser abordado desde a primeira série do Ensino Médio, tentando aplicar aos diversos assuntos que permeiam o conteúdo programático do Ensino Médio.

A sequência didática se dará em 8 (oito) aulas de cinquenta minutos cada, segundo a seguinte tabela:

Tabela 6 – Sequência Didática Proposta

Aula 1	Software SageMath
Aula 2	Python e seus Comandos Básicos
Aula 3	Primeiras Noções / Grafos
Aula 4	Representando um Grafo por uma Matriz / Subgrafos
Aula 5	Grau de um Vértice
Aula 6	Caminhos e Grafos Conexos / Ciclos
Aula 7	Grafos Eulerianos e Hamiltonianos / O Problema do Menor Caminho
Aula 8	Árvores e Florestas / O Problema de Conexão de Peso Mínimo

As atividades propostas em cada aula devem ser discutidas e solucionadas, de preferência, em grupos de cinco alunos, para que possam compartilhar suas ideias e dúvidas com os outros colegas e juntos chegarem a uma solução para os problemas.

Na Seção 5.1 a seguir, serão abordadas cada uma das aulas e as atividades propostas, descrevendo os objetivos, os conteúdos trabalhados, a metodologia e o instrumento de avaliação em cada etapa. As soluções das atividades propostas seguem na Seção 5.2.

5.1 SEQUÊNCIA DIDÁTICA

Para a realização de todas as aulas serão necessários que os alunos estejam com computadores ou *notebook*'s ou *tablet*'s ou *smartphones* com internet à disposição para a utilização online do *software* SageMath.

5.1.1 Aula 1

5.1.1.1 Objetivos

- Reconhecer e utilizar o *software* SageMath como aliado na resolução de problemas.

5.1.1.2 Conteúdos trabalhados

- O que é o *Software* SageMath?
- Como se cadastrar no CoCalc (SageMath *online*)?

5.1.1.3 Metodologia

Essa primeira aula será para os alunos se adaptarem ao ambiente do SageMath.

O professor fará a apresentação do que se trata o *software* SageMath e em seguida, mostrará como se cadastrar no CoCalc, a plataforma de acesso ao SageMath na nuvem.

Após os alunos terem feito o cadastro no CoCalc e criado o próprio projeto, o professor mostrará algumas ferramentas de uso do SageMath, conforme exemplos apresentados na Seção 1.4 sobre o ambiente SageMath e suas funcionalidades básicas.

5.1.1.4 Instrumento de avaliação

Nessa primeira aula, não avaliaremos o aluno de forma efetiva, pois se trata de conhecer e se cadastrar no ambiente do CoCalc.

5.1.2 Aula 2

5.1.2.1 Objetivos

- Ter um primeiro contato com os comandos básicos de lógica de programação usando a linguagem Python e compreender o funcionamento de cada comando.

5.1.2.2 Conteúdos trabalhados

- O que é Python e por que usá-lo?
- Comandos básicos de lógica de programação usando a linguagem Python.

5.1.2.3 Metodologia

O professor apresentará comandos de lógica de programação usando Python, onde serão apresentados os códigos relativos à função do Algoritmo da Divisão Euclidiana e à função da soma dos termos da PA.

Após essa sequência, propõe-se que sejam trabalhadas em sala as seguintes atividades:

Atividade 1: [Números Perfeitos] Um número n se diz "**PERFEITO**" se é igual à soma de seus divisores, sem contar com o próprio número.

Por exemplo, os divisores do número 6 são 1, 2, 3 e 6, e excluindo o próprio número (6), temos a soma igual a 6 ($1 + 2 + 3 = 6$).

Escreva um programa que retorne todos os números perfeitos até 10000, de preferência peça para imprimir os divisores próprios do número perfeito.

Atividade 2: [Raízes Reais de uma Função Quadrática] Dada a função $f(x) = ax^2 + bx + c$, existirão três casos a serem considerados para a obtenção do número de raízes reais da função quadrática. Isso dependerá do valor do discriminante Δ.

1º caso $\Rightarrow \Delta > 0$: A função possui duas raízes reais e distintas, isto é, diferentes.

2º caso $\Rightarrow \Delta = 0$: A função possui raízes reais e iguais. Nesse caso, dizemos que a função possui uma única raiz.

3º caso $\Rightarrow \Delta < 0$: A função não possui raízes reais.

Escreva uma função cujas entradas sejam os coeficientes a, b e c e que retorne a informação se essa função possui raízes reais e em caso positivo, quantas e quais são as raízes reais.

5.1.2.4 Instrumento de avaliação

A avaliação do aprendizado dos alunos será feita a partir da atividade trabalhada, nos momentos em que cada grupo expõe seus resultados, observando se o código funciona bem, e em caso de erros, os alunos da turma juntamente com o professor tentarão corrigir o erro. Dessa forma, o professor observará os pontos onde apareceram mais dúvidas.

5.1.3 Aula 3

5.1.3.1 Objetivos

- Estimular o aluno a usar a representação de grafos sem mesmo conhecer a definição como forma de melhor visualização para a solução de um problema.
- Definir formalmente um grafo e conhecer outras definições básicas dessa teoria.
- Conhecer como declarar um grafo no SageMath e obter informações desses grafos a partir de comando do SageMath.

5.1.3.2 Conteúdos trabalhados

- O que é um Grafo e como o definir?
- Conhecer os conceitos básicos de grafos simples, laços, arestas múltiplas e grafos completos.
- Fazer uso da programação para obter resultados que não sejam acessíveis diretamente.

5.1.3.3 Metodologia

O professor iniciará a aula apresentando o problema dos amigos (Exemplo 2 do Capítulo 2) e mostrando como o representa usando grafos e em seguida, define matematicamente o que é um grafo.

Depois dos primeiros contatos com a ideia de grafos, o professor mostrará como definir um grafo no SageMath e como visualizá-lo, e para isso, deverá ser apresentado as formas de definir um grafo sequencialmente conforme apresentado no Capítulo 2 - Seção 2.1, até que o aluno conheça a forma principal de definir um grafo no SageMath.

Em seguida, é proposta a primeira atividade da aula, a fim de introduzir os conceitos laços, arestas paralelas e grafos simples. O resultado dessa primeira atividade será visualizar o grafo do Exemplo 3, apresentado na Seção 2.2.

Atividade 1: Considere um grafo G definido matematicamente por

G = {{A,{A, E, F}}, {B,{C, D, E, E, E, G}}, {C,{C, F, G}}, {D,{G, G}},
{E,{A, B, B, B}}, {F, {A, C, G}}, {G, {B, C, D, D, F}}}

Defina esse grafo G no SageMath e visualize esse grafo.

Após apresentar os conceitos acima, o professor apresenta o comando **.has_loops()** e em seguida, o comando **remove_loops()**.

Terminada essa etapa, o professor deverá apresentar o exemplo sobre as fronteiras dos estados do Nordeste brasileiro (Exemplo 4), a fim de apresentar os conceitos de ordem e tamanho de um grafo, construindo o grafo desse exemplo no SageMath; bem como apresentando o comando para obter a ordem e o tamanho nesse ambiente.

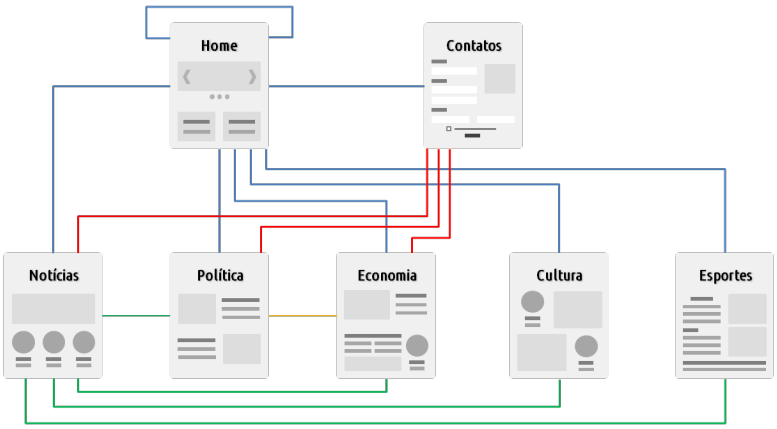
Na sequência, o professor conceituará grafo completo e mostrará como obter um grafo completo com k vértices. Por fim, são propostos as atividades a seguir.

Atividade 2: [Conexões entre Páginas WEB] Um programador ao elaborar um *site* precisa fazer as conexões entre a barra de navegação (menu) do *site* com as páginas disponíveis. Ele pretende criar esse menu conforme a figura abaixo:

Figura 42 – Menu da página inicial do *site* criado pelo programador

Porém, algumas dessas paginas não apresentam todos os *link*'s desse menu e, dessa forma, só conseguem acessar determinadas páginas ao retornar para a página inicial do site, denominada HOME, ou para outras em que existam algum tipo de conexão com aquelas páginas que se desejam acessar. Na figura a seguir, estão representadas as páginas de um site com as devidas relações com outras páginas.

Figura 43 – Ligações entre páginas do *site*



Observe que mesmo estando na página inicial (HOME), um dos *link*'s da página inicial remete a própria página, por isso existe a ligação da HOME para a HOME.

Sobre o problema acima, faça o que se pede:

(a) Represente essa situação por um grafo.

(b) Esse grafo é simples?

(c) Defina esse grafo no SageMath e visualize-o.

(d) Estando na página de POLÍTICA, é possível acessar diretamente a página CULTURA?

(e) Estando na página de ESPORTES, é possível acessar diretamente a página NOTÍCIAS?

Atividade 3: [Campeonato Pernambucano - Hexagonal do Título] [Adaptado da reportagem do *site* CoralNET] "O fim da primeira fase do Campeonato Pernambucano 2017 definiu o Hexagonal do Título. Salgueiro, Belo Jardim e Central, respectivamente, 1º, 2º e 3º colocados na primeira fase, se juntam ao trio de ferro - Santa Cruz, Sport Recife

e Náutico. O regulamento é o mesmo das últimas edições do estadual. As seis equipes se enfrentam entre si em jogos de ida e volta e os quatro melhores colocados avançando ao mata-mata, com semifinal ocorrendo em dois dias e a final também."

Sobre a situação acima:

(a) Represente a situação acima usando a ideia de grafos de modo que a aresta conectando dois times indicam a existência de dois confrontos entre essas duas equipes.

(b) Esse grafo é simples? É completo?

(c) Defina esse grafo no SageMath e visualize-o.

(d) Qual a ordem e o tamanho desse grafo?

5.1.3.4 Instrumento de avaliação

A avaliação do aprendizado dos alunos será feita da mesma forma que a primeira aula baseada na atividade trabalhada, nos momentos em que cada grupo expõe seus resultados, observando se o código atinge o resultado solicitado, e em caso de erros, a correção em conjunto constituirá uma ferramenta de aprendizado. Dessa forma, o professor observará os pontos onde apareceram mais dúvidas, reforçando tais tópicos.

5.1.4 Aula 4

5.1.4.1 Objetivos

- Apresentar outras formas de representação de grafos usando matrizes.
- Usar a programação para alcançar resultados desejados.

5.1.4.2 Conteúdos trabalhados

- Definição de matriz de incidência e matriz de adjacência de um grafo.
- Definição de subgrafos.
- Definição de complemento de um grafo.

5.1.4.3 Metodologia

O professor iniciará a aula apresentando o grafo G (Exemplo 6 do Capítulo 2) e mostrando que esse grafo pode ser representado por meio de matrizes, introduzindo os conceitos de matriz de incidência e de adjacência.

Na construção da matriz de incidência e fazendo uso do grafo G apresentado anteriormente, o professor deve lançar ao aluno como curiosidade o trecho dessa dissertação na parte que trata de matriz de incidência, a saber: "Curiosamente, a soma dos valores de qualquer coluna em uma matriz de incidência é sempre 2. Notou isso? E a explicação, será que você consegue perceber o porquê?".

Da mesma forma, na construção da matriz de adjacência deve ser feito o mesmo com a curiosidade proposta no trecho "O que a soma das colunas ou a soma das linhas de uma matriz de adjacência de um grafo G representa?"

É essencial aqui que, na apresentação dessas curiosidades, não sejam introduzidos conceitos que serão tratados posteriormente, falando apenas no número de vértices que uma aresta atinge na primeira curiosidade e sobre o número de arestas que atingem um vértice.

Na sequência, deve ser apresentado como obter essas matrizes no SageMath e qual a forma de organização de vértices e arestas feita pelo SageMath.

Depois de apresentada a representação de um grafo usando matrizes, o professor deve propor aos alunos a atividade a seguir para que obtenham o grafo relativo à matriz de adjacência proposta, além de obter a matriz de incidência desse grafo.

Após tal atividade, o professor mostrará como obter o grafo a partir da matriz de adjacência apresentada fazendo o uso do SageMath.

Por fim, o professor introduz o conceito de subgrafo de forma simples e deve propor a atividade proposta a seguir.

Atividade 1: [Obtendo o grafo a partir de uma matriz de adjacência] Considere a matriz de adjacência de um grafo G, apresentada a seguir.

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

(a) Esse grafo é simples?

(b) Represente essa situação por um grafo.

(c) Obtenha a matriz de incidência desse grafo G.

(d) Obtenha o grafo a partir dessa matriz no SageMath.

(e) Obtenha a matriz de incidência desse grafo G no SageMath.

(f) Obtenha três subgrafos desse grafo G.

5.1.4.4 Instrumento de avaliação

A avaliação do aprendizado seguirá as propostas anteriores e será feita a partir da atividade trabalhada, nos momentos em que cada grupo apresenta seus resultados, observando se o código funciona bem, e em caso de mal funcionamento do código, os alunos da turma juntamente com o professor tentarão corrigir o erro. Dessa forma, o professor observará os pontos onde aparecerão mais dúvidas.

5.1.5 **Aula 5**

5.1.5.1 Objetivos

- Conhecer o conceito de grau de um vértice que é uma ferramenta essencial na construção de outros conceitos mais avançados de grafos.
- Usar a programação para alcançar resultados desejados.

5.1.5.2 Conteúdos trabalhados

- Conceito de grau de um vértice e sua importância no estudo de grafos.
- Definição de complemento de um grafo.

5.1.5.3 Metodologia

O professor iniciará a aula introduzindo o conceito de grau de um vértice. Para isso, deve fazer uso do grafo G do Exemplo 6 do Capítulo 2 e do grafo G obtido na atividade anterior. O professor deve propor que os alunos somem todos os graus em cada grafo e que pensem o porquê dessas somas serem sempre números pares.

Em seguida, deve ser apresentado o Teorema 7 sobre a soma dos graus de todos os vértices em qualquer grafo ser sempre par e igual ao dobro do número de arestas. Também deve ser apresentada a consequência desse Teorema, o Colorário 8, popularmente conhecido como Lema do aperto de mãos.

Depois de conhecer o conceito de grau, o professor deve mostrar como obter o grau dos vértices das diversas formas no SageMath, bem como o SageMath ordena a sequência de graus.

Após essa sequência, propõe-se que sejam trabalhadas em sala as seguintes atividades:

Atividade 1: [Potência 2 da Matriz de Adjacência] Considere um grafo G simples (grafo que não possui laços nem arestas paralelas) e seja A a matriz de adjacência desse grafo G.

(a) Mostre que a diagonal principal da matriz A^2 equivale à sequência de graus dos vértices do grafo G.

(b) Implemente uma função no SageMath que receba um grafo X e retorne o(s) vértice(s) de maior grau.
DICA: *Use a diagonal principal da matriz A^2 para obter a sequência de graus do grafo X.*

Atividade 2: [Complemento de um Grafo] Seja G um grafo simples. O complemento $\overline{G}$ de G é um grafo simples em que o conjunto dos vértices é V(G) e cujas arestas são os pares de vértices não adjacentes de G.

(a) Implemente uma função no SageMath que receba um grafo X e imprima o seu complemento.
DICA: *Use a ideia de grafo completo, complete uma cópia do grafo adicionando as arestas que faltam e retire as arestas que fazem parte do grafo X.*

(b) Expresse a sequência de graus de $\overline{G}$ em termos da sequência de graus de G.

(c) Existe um comando no SageMath que apresenta o complemento de um grafo G digitando **G.complement()**. Use esse comando e visualize seu grafo digitando **show(G.complement())**. O resultado da letra (a) corresponde ao que foi apresentado usando o comando **G.complement()**?

5.1.5.4 Instrumento de avaliação

A avaliação do aprendizado seguirá as propostas anteriores e será feita a partir da atividade trabalhada, nos momentos em que cada grupo apresenta seus resultados, observando se o código funciona bem, e em caso de mal funcionamento do código, os alunos da turma juntamente com o professor tentarão corrigir o erro. Dessa forma, o professor observará os pontos onde aparecerão mais dúvidas.

5.1.6 Aula 6

5.1.6.1 Objetivos

- Conhecer os conceitos de passeio, trilha e caminho em um grafo.

- Entender a ideia de um grafo ser conexo ou desconexo.
- Compreender os conceitos de grafo bipartido e de ciclo, bem como a relação existente entre esses dois conceitos.
- Usar a programação para alcançar resultados desejados.

5.1.6.2 Conteúdos trabalhados

- Definição de passeio, trilha e caminho de um grafo.
- Definição de menor caminho.
- Conceito de grafo conexo e de grafo desconexo.
- Definição de grafo bipartido.
- Definição de ciclo.
- Relação entre grafo bipartido e ciclo ímpar.

5.1.6.3 Metodologia

O professor iniciará a aula retomando o grafo G do problema dos amigos (Exemplo 2 do Capítulo 2) e a partir desse grafo, introduzirá os conceitos de passeio, comprimento de um passeio, trilha e caminhos, bem como o conceito de menor caminho.

Em seguida, o professor deve mostrar como obter, no SageMath, os conceitos abordados acima a partir do grafo retomado no inicio da aula.

Na sequência, devem ser apresentados os conceitos de componentes e de grafos conexos e desconexos, bem como obter se um grafo é conexo ou não, usando o SageMath.

Usando o mesmo grafo do Exemplo 2, o professor deve apresentar aos alunos, de forma sucinta, a análise interessante que gira em torno de uma potência k da matriz de adjacência de um grafo G $[\mathbf{A(G)}]$, envolvendo o conceito de passeios nesse grafo.

Em seguida, o professor deve introduzir o conceito de grafos bipartidos, usando o Exemplo 10 apresentado na Seção 3.1, seguindo a sequência apresentada. Depois deve ser mostrado como saber se um grafo é bipartido ou não, e em caso positivo, como obter essa bipartição, através do SageMath.

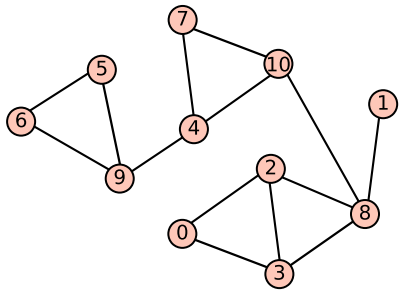
Após a introdução do conceito de grafo bipartido, o professor deve introduzir o conceito de ciclos e de grafos ciclo, destacando a noção de ciclo par e ciclo ímpar, além dos conceitos de grafo ciclico e acíclico, bem como obter essas informações no SageMath.

Em seguida, deve ser apresentado o Teorema 11 sobre a relação entre ciclos e grafos bipartidos, assim como apresentar uma forma de testar se um grafo é bipartido usando a ideia de paridade das distâncias.

Após essa sequência, são propostas as seguintes atividades para que sejam trabalhadas em sala:

Atividade 1: Considere o grafo G a seguir.

Figura 44 – Grafo G



(a) Apresente exemplos de passeio, trilha, caminho e ciclo no grafo G.

(b) Encontre o menor caminho conectando os vértices 0 e 5.

(c) Declare esse grafo G no SageMath.

(d) Obtenha todos os menores caminhos que partem do vértice 0 usando o SageMath.

(e) Define-se como **ponte** (em inglês, *bridge*) ou **aresta de corte** (em inglês, *cut edge*) a uma aresta que se removida do grafo, aumenta o número de componentes conexos. De outra forma, uma aresta é uma ponte, se e somente se, ela não está contida em qualquer ciclo. Encontre as arestas do grafo G que são pontes.

(f) No SageMath é possível obter a lista das pontes de um grafo G usando o comando **G.bridges()**. Use esse comando no SageMath para conferir o resultado obtido no item (e).

(g) Implemente uma função no SageMath que receba um grafo X e retorne uma lista com as pontes desse grafo.
DICA: *Use o comando* ***X.connected_components_number()*** *para calcular o número de componentes conexos do grafo X.*

(h) Define-se como **vértice de corte** (em inglês, *cut edge*) ou **ponto de articulação** a um vértice que se removido do grafo, aumenta o número de componentes conexos. Encontre os vértices do grafo G que são pontos de articulação.

(i) Implemente uma função no SageMath que receba um grafo X e retorne uma lista com os vértices de corte desse grafo.
DICA: *Use a mesma dica do item (g).*

Atividade 2: Uma pequena fábrica tem cinco máquinas - 1, 2, 3, 4 e 5 - e seis operários - A, B, C, D, E e F. A tabela especifica as máquinas que cada operário sabe operar:

A	2, 3
B	1, 2, 3, 4, 5
C	3

D	
E	2, 4, 5
F	2, 5

(a) O grafo formado pelas arestas que conectam os operários com as máquinas que sabem operar é um grafo bipartido?

(b) Represente essa situação por um grafo.

(c) Esse grafo é conexo?

(d) Obtenha o grafo no SageMath.

(e) Obtenha a matriz de adjacência desse grafo no SageMath. Que aparência tem a matriz de adjacências de um grafo bipartido?

(f) A **matriz da bipartição** de um grafo {U,W}-bipartido é definida assim: cada linha da matriz é um elemento de U, cada coluna da matriz é um elemento de W e no cruzamento da linha u com a coluna w temos um 1 se uw é uma aresta e temos um 0 em caso contrário. Escreva a matriz da bipartição do grafo dessa atividade. Adote a bipartição óbvia: U = {A, B, C, D, E, F} e W = {1, 2, 3, 4, 5}.

(g) Implemente uma função no SageMath que receba um grafo X e retorne as bipartições e a matriz de bipartição relacionada, se o grafo for bipartido; caso contrário, informe que o grafo não é bipartido.

5.1.6.4 Instrumento de avaliação

A avaliação do aprendizado continuará a ser feita conforme propostas anteriores e será baseada a partir da atividade trabalhada, nas situações em que cada grupo apresenta seus resultados, observando se o código roda bem, e em caso de falha no funcionamento do código, os alunos juntamente com o professor tentarão corrigir o erro. Dessa forma, o professor reforçará os pontos onde surgirem mais dúvidas.

5.1.7 Aula 7

5.1.7.1 Objetivos

- Conhecer os conceitos de grafo euleriano e de grafo hamiltoniano.
- Conhecer o algoritmo de Dijkstra e o problema do menor caminho.
- Fazer uso de programação para alcançar resultados desejados.

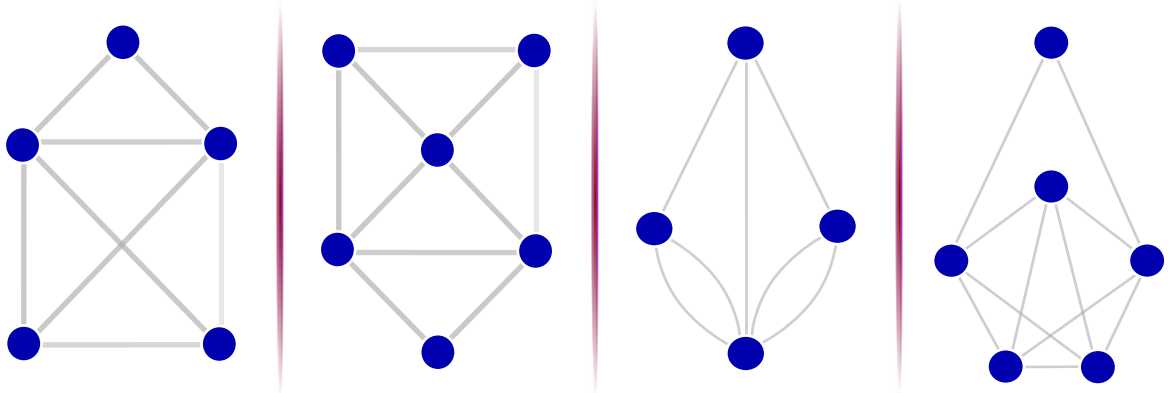
5.1.7.2 Conteúdos trabalhados

- Definição de circuito euleriano, caminho euleriano e de grafo euleriano.
- Relação entre grafo euleriano e o grau dos vértices.
- Definição de caminho hamiltoniano e de grafo hamiltoniano.
- O problema do menor caminho e o algoritmo de Dijkstra.

5.1.7.3 Metodologia

O professor iniciará a aula apresentando o conceito de circuito euleriano, de grafo euleriano e de caminho euleriano. Em seguida, é proposta a atividade a seguir, conforme é apresentado na Seção 3.3 do Capítulo 3.

Atividade: É possível sair de um vértice e percorrer todas as arestas sem repetir nenhuma delas, retornando ao mesmo vértice inicial? De uma maneira mais prática, é possível desenhar esses grafos sem retirar a ponta do lápis do papel, retornando ao vértice de início?



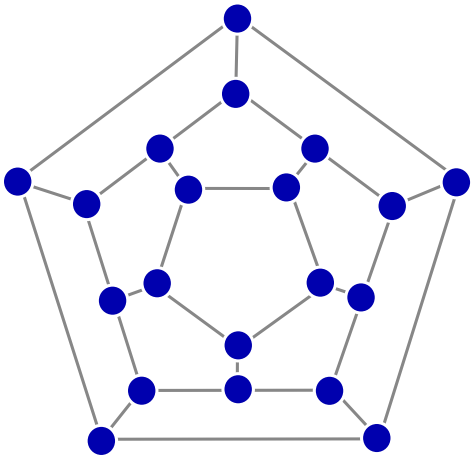
Após as tentativas dos alunos, o professor ainda não deve dizer quais grafos da figura acima é possível fazer um circuito euleriano. Em seguida, deve ser apresentado o

Teorema de Euler (Teorema 13), a ser usado como ferramenta para a solução da atividade apresentada, assim como o conceito de grafo semieuleriano e o colorário 14.

Apresentados esses conceitos e teoremas, o professor corrige a atividade com a turma, tentando associar com tais conceitos e teoremas.

Na sequência, deve ser apresentado os conceitos de caminho hamiltoniano, ciclo hamiltoniano, grafo hamiltoniano e grafo semi-hamiltoniano.

Como exemplo para abordagem dos conceitos acima, o professor deve usar o jogo inventado por Hamilton, que envolve um dodecaedro, apresentado na figura 24, cujo objetivo do jogo era que o jogador viajasse "ao redor do mundo" determinando uma viagem circular que incluísse todas as cidades exatamente uma vez, com a restrição de que só fosse possível viajar de uma cidade a outra se existisse uma aresta entre os vértices correspondentes.

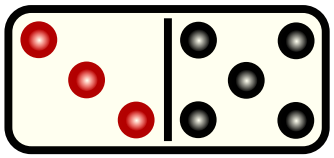


Em seguida, o professor deve mostrar como obter as informações apresentadas acima envolvendo grafos eulerianos e hamiltonianos usando o SageMath, em especial se um grafo é ou não euleriano, se um grafo é ou não hamiltoniano, circuitos eulerianos, ciclos hamiltonianos e caminhos hamiltonianos.

Depois de apresentados esses conceitos, o professor deve propor aos alunos a atividade a seguir.

Atividade 1: [Problema do Dominó] Imagine que consideramos os valores que aparecem no dominó como vértices e as peças do dominó como arestas que conectam esses vértices.

Por exemplo, a peça "terno e quina" abaixo conecta os vértices 3 e 5.



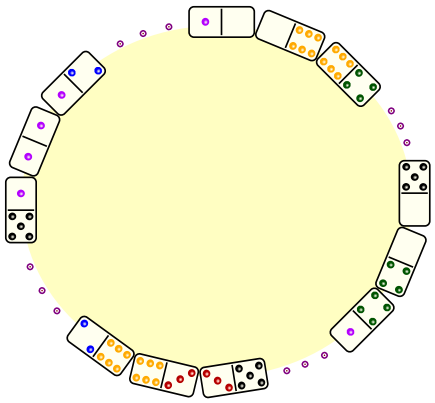
(a) Esse grafo é simples?

(b) Represente essa situação usando grafo.

(c) Esse grafo é completo? Removido os laços desse grafo, obtemos um grafo completo?

(d) Obtenha esse grafo no SageMath criando um código que adicione essas arestas usando laços repetitivos. Ao declarar o grafo vazio, é importante permitir laços digitando **G = Graph(loops=True)**.

(e) É possível com as pedras de um jogo de dominó formar um anel (seguindo as regras do jogo) como mostra a figura a seguir?



(f) Suponha agora um dominó diferente com valores de 0 a 7. Nesse novo dominó é possível com as pedras desse jogo formar um anel, seguindo as regras do jogo, como mostrado na figura anterior?

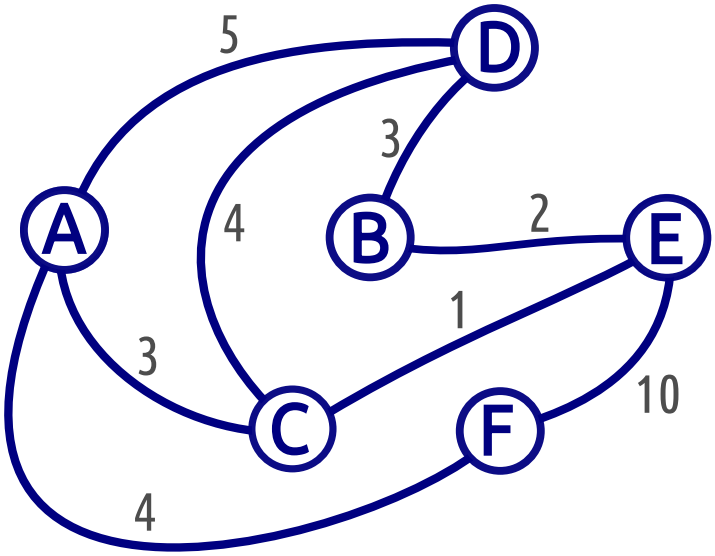
Após essa atividade do problema do dominó, o professor deve introduzir as ideias que envolvem os conceitos de grafo valorado, peso de uma aresta e problema do menor caminho, fazendo uso do problema do menor caminho (Exemplo 17 da Seção 3.4).

Para tal, deve-se introduzir o algoritmo de Dijkstra e resolver esse problema conforme solução mostrada nessa Seção. Ao término, o professor deve mostrar como declarar um grafo valorado e como adicionar as arestas com seus respectivos pesos no SageMath, bem como conferir o resultado obtido para o menor caminho do Exemplo 17 usando o comando **G.shortest_paths('A', by_weight=True)**.

Após essa sequência, propõe-se que seja trabalhada em sala a seguinte atividade:

Atividade 2: Considere o grafo valorado a seguir.

Figura 45 – Grafo valorado



Qual o menor caminho (com peso minímo) conectando todos os vértices desse grafo partindo de D?

5.1.7.4 Instrumento de avaliação

A avaliação do aprendizado seguirá as propostas anteriores e será feita a partir da atividade trabalhada, nos momentos em que cada grupo apresenta seus resultados, observando se o código funciona bem, e em caso de mal funcionamento do código, os alunos da turma juntamente com o professor tentarão corrigir o erro. Dessa forma, o professor observará os pontos onde aparecerão mais dúvidas.

5.1.8 **Aula 8**

5.1.8.1 Objetivos

- Conhecer os conceitos de árvore e floresta.
- Compreender as particularidades do conceito de árvore, relacionando com o número de vértices e arestas e o grau dos vértices.
- Conceituar árvore geradora e conhecer algoritmos que permitam obter uma árvore geradora.
- Aplicar algoritmos que permitam obter uma solução para o problema de conexão de custo mínimo.
- Usar a programação como aliado na busca de resultados esperados.

5.1.8.2 Conteúdos trabalhados

- Definição de árvore e floresta.
- Relação entre árvore e o número de arestas e vértices.
- Relação entre árvore e o grau dos vértices que a compõe.
- Definição de árvore geradora.
- Apresentação dos algoritmos de busca em profundidade e de busca em largura.
- Apresentação dos algoritmos de Prim e de Kruskal na solução do problema de conexão de peso mínimo.

5.1.8.3 Metodologia

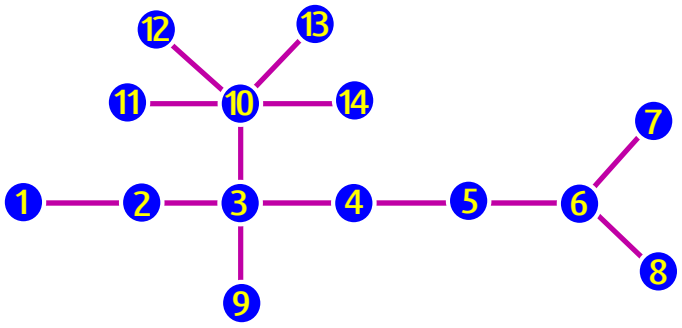
O professor iniciará a aula introduzindo os conceitos de árvore e floresta e apresentando exemplos de tais grafos. Em seguida, o professor deve apresentar o Teorema 18 sobre o caminho único conectando dois vértices quaisquer em uma árvore, sendo fundamental que seja feita uma prova intuitiva desse Teorema.

Na sequência, deve-se introduzir o Teorema 19, mostrando a relação entre o número de vértices e arestas em uma árvore, bem como o Colorário 20, consequência do teorema anterior, sobre o número de vértices de grau 1 em uma árvore não trivial.

Conhecidos esses teoremas, o professor deve apresentar os comandos do SageMath que permitem reconhecer se um grafo é uma árvore (**G.is_tree()**), além de como obter todas as árvores a partir de um número n de vértices e como visualizá-los.

Em seguida, o professor pode propor a atividade a seguir.

Atividade 1: Considere um grupo de 14 pessoas. Suponha que a comunicação entre as pessoas deste grupo esteja representada através do grafo a seguir, onde os vértices representam as pessoas e as arestas representam as ligações entre as pessoas.

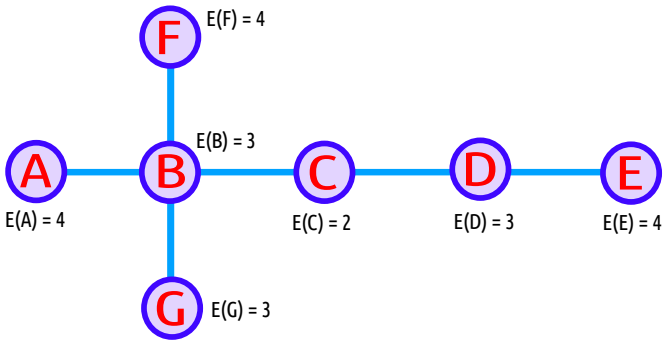


(a) Esse grafo é uma árvore?

(b) Uma folha de uma árvore ou floresta é qualquer vértice de grau 1. Determine todas as folhas do grafo acima.

(c) Como o grafo é conexo, sabemos que todos os membros podem ser alcançados diretamente ou através de outros membros do grupo. É possível a comunicação direta entre as pessoas 6 e 11? Determine uma sequência de forma que a pessoa 13 possa se comunicar com a pessoa 8. Existe mais de uma forma de comunicação entre duas pessoas quaisquer?

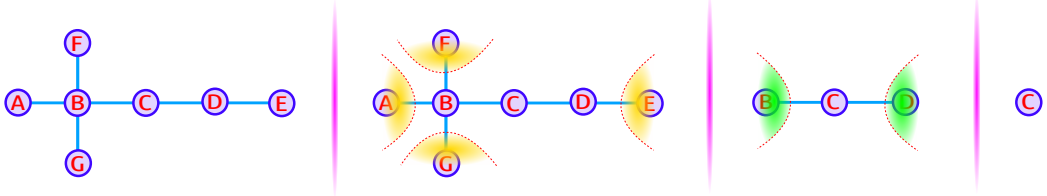
(d) Definimos a excentricidade de um vértice v, $E(v)$, como sendo o valor da distância máxima entre v e w, para todo $w \in V$. Lembre-se que a distância entre dois vértices v e w é o menor caminho entre v e w. Por exemplo, para o grafo abaixo, são apresentadas as excentricidades de cada vértice.



Define-se como **centro de um grafo** o(s) vértice(s) (pode existir mais de um) que tem o menor valor de excentricidade. Nesse caso, o centro é o vértice C.

Para o caso específico das árvores, uma maneira de determinar o **centro de uma árvore** é eliminando progressivamente os vértices pendentes (folhas) e as arestas incidentes a eles até que reste um vértices isolado (o centro) ou dois vértices ligados por uma aresta (o bicentro). Como dito, essa maneira só se aplica em árvores; para outros grafos, deve-se analisar a excentricidade de cada vértice e usar a definição de centro de um grafo. Vejamos a obtenção do centro da árvore apresentada no exemplo.

Eliminamos as folhas a cada passo até sobrar um ou dois vertices centrais!



Observe que ao retirarmos um vértice de um grafo, retiramos também uma aresta. Assim, o grau dos vértices que permanecem no grafo pode diminuir de valor e a excentricidade do vértice que permanece no grafo diminui de valor.

Obtenha o centro da árvore que representa a comunicação entre as pessoas do grupo, mostrada no início dessa atividade, ou seja, num contexto aplicado, se for usado o critério de facilidade de acesso às pessoas quem deverá ser escolhido como líder de grupo?

(e) Declare o grafo dessa atividade no SageMath e visualize-o.

(f) É possível encontrar o centro de um grafo G usando o comando **G.center()** no SageMath. Encontre o centro do grafo declarado e compare com o resultado obtido no item (d).

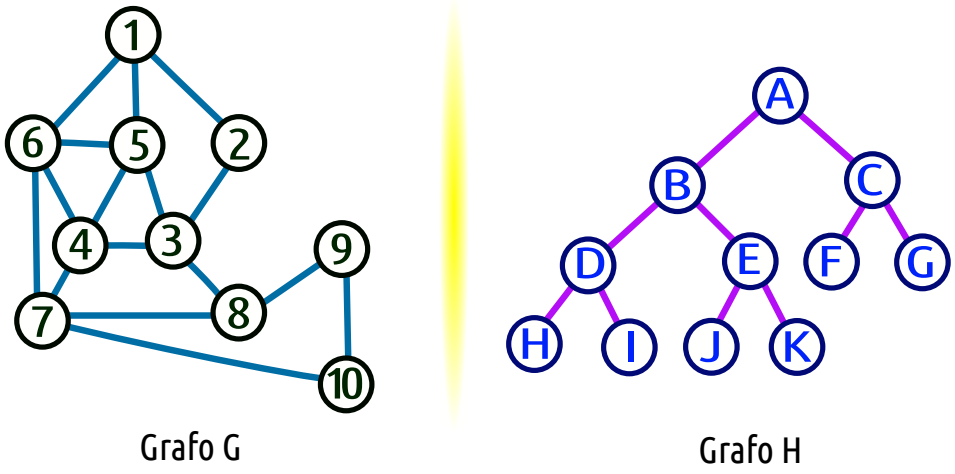
(g) Implemente uma função no SageMath que receba um grafo X, analise se X é uma árvore e retorne o centro de X a partir da matriz de distância (**X.distance_matrix()**).

Após essa Atividade 1, o professor deve introduzir o conceito de árvore geradora de um grafo e como encontrá-la em um grafo, abordando os algoritmos de busca em profundidade e de busca em largura.

Depois de apresentados esses algoritmos para obtenção de uma árvore geradora, deve-se mostrar como obter uma árvore geradora no SageMath e outros comandos relacionados.

Em seguida, é proposta a atividade seguinte para obtenção de uma árvore geradora usando um dos algortimos apresentados.

Atividade 2: Considere os grafos G e H abaixo.



Grafo G Grafo H

(a) Qual desses grafos é uma árvore?

(b) Faça uma busca em largura e uma busca em profundidade nos grafos acima para obter uma árvore geradora.

(c) Qual o centro do grafo H? E do grafo G?

(d) Declare os grafos dessa atividade no SageMath e visualize-os.

(e) Use a função que você criou no item (g) da atividade 1 dessa aula e verifique se o que você obteve nos itens (a) e (c) corresponde ao que foi retornado pela função.

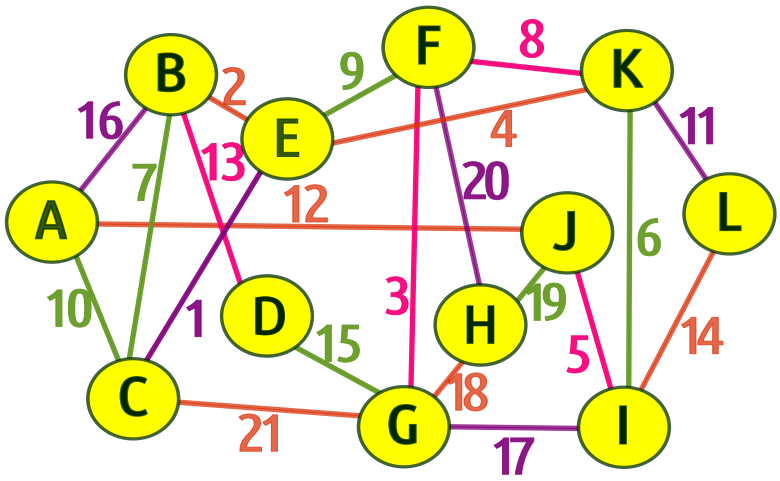
Após a Atividade 2, o professor deve apresentar o problema da instalação da tubulação de condicionador de ar (Exemplo 21 da Seção 4.2) como motivação inicial para o problema de conexão de custo mínimo e discutir, junto aos alunos, sua solução, apresentando os algoritmos de Kruskal e de Prim, conforme detalhado nessa Seção.

Em seguida, o professor apresenta como obter a árvore geradora de custo mínimo no SageMath por meio de cada algoritmo (Kruskal e Prim) a partir do comando **G.min_spanning_tree()**.

Como proposta auxiliar, o professor pode propôr a construção coletiva da função para obter uma árvore geradora máxima (de custo máximo), conforme apresentado no final dessa Seção.

Após essa sequência, propõe-se que seja trabalhada em sala a seguinte atividade:

Atividade 3: [Árvore Geradora] Considere o grafo valorado abaixo.



(a) Utilize os algoritmos de Kruskal e de Prim para identificar uma árvore geradora mínima. Adapte os algoritmos para obter uma árvore geradora máxima.

(b) Qual o peso da árvore geradora mínima e máxima que representa o grafo da figura acima? O peso é a soma dos valores das arestas da árvore resultante.

5.1.8.4 Instrumento de avaliação

A forma de avaliação continuará seguindo as ideias anteriores e será mensurada a partir da atividade trabalhada, nas apresentações dos resultados de cada grupo, observando o funcionamento correto do código, e em caso de falha na execução do código, os alunos juntamente com o professor tentará corrigir o erro. Dessa forma, o professor analisará os pontos onde apareceram mais dúvidas e tentará dar um parecer para a turma, de forma a não retornarem ao erro.

5.2 SOLUÇÃO PARA AS ATIVIDADES PROPOSTAS

5.2.1 **Aula 1**

Não há atividades propostas nessa aula.

5.2.2 **Aula 2**

Atividade 1 [Números Perfeitos]

UMA SOLUÇÃO:

Código 5.1 – Algoritmo para encontrar os Números Perfeitos até 10000

```
1  n = 1
2  while n != 10000:
3      soma = 0
4      for i in range(1,n):
5          if n % i == 0:
6              soma = soma + i
7
8      if soma == n:
9          print n, "é um número perfeito!"
10
11     n = n + 1
```

Resultado apresentado para o código acima:

```
6 é um número perfeito!
28 é um número perfeito!
496 é um número perfeito!
8128 é um número perfeito!
```

Atividade 2 [Raízes Reais de uma Função Quadrática]

UMA SOLUÇÃO:

Código 5.2 – Algoritmo para encontrar as Raízes Reais de uma Função Quadrática

```
def raizesfquad(a,b,c):
    delta = b^2 - 4*a*c

    if delta < 0:
        print "Não há raízes reais!"
    elif delta == 0:
        x = -b/(2*a)
        print "Existe uma raiz real única:", x
    else:
        x1 = (-b - sqrt(delta))/(2*a)
        x2 = (-b + sqrt(delta))/(2*a)
        print "Existem duas raízes reais distintas:", x1, "e", x2
```

Testando o código acima para as funções quadráticas e seus respectivos comandos $f(x) = x^2 - 5x + 6$ (**raizesfquad(1,-5,6)**), $f(x) = x^2 + x + 1$ (**raizesfquad(1,1,1)**) e $f(x) = 2x^2 + 12x + 18$ (**raizesfquad(2,12,18)**), são apresentados os resultados a seguir:

```
Existem duas raízes reais distintas: 2 e 3
Não há raízes reais!
Existe uma raiz real única: -3
```

5.2.3 Aula 3

Atividade 1

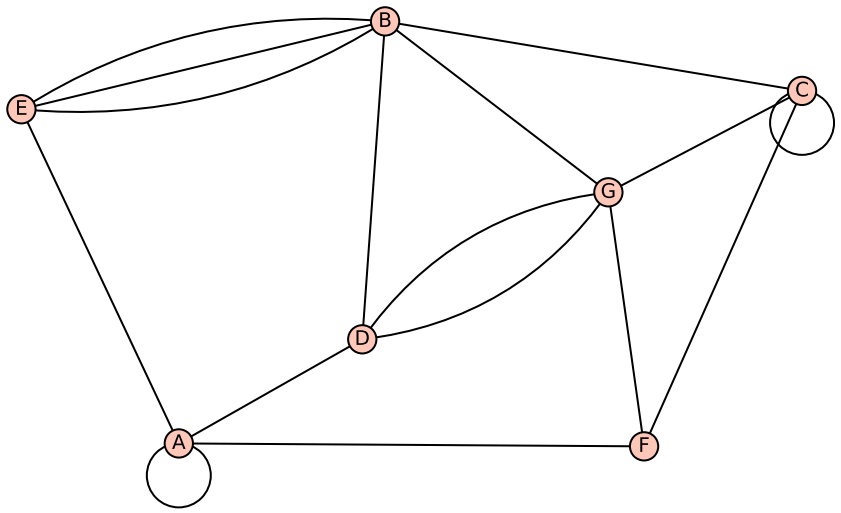
UMA SOLUÇÃO:

Código 5.3 – Algoritmo para obter o grafo G da atividade 1

```
G = Graph({'A':['A', 'E', 'F'], 'B': ['C', 'D', 'E', 'E', 'E', 'G'], 'C
    ': ['C', 'F', 'G'], 'D': ['A', 'B', 'B', 'B'], 'F': ['A', 'C','G'], '
    G': ['B', 'C', 'D', 'D', 'F']})

plot(G)
```

Resultado apresentado para o código acima:



Atividade 2 [Conexões entre Páginas WEB]

UMA SOLUÇÃO:

(a) A cargo do aluno.

(b) Não é simples, pois contém laço (vértice HO).

(c) Podemos representar essa situação utilizando conceitos da teoria dos grafos, nomeando os vértices pelo nome de cada página (aqui usaremos as duas primeiras letras de cada nome; por exemplo, para a página Economia usaremos EC), as linhas conectando duas páginas indicam a existência de *link* ligando essas páginas.

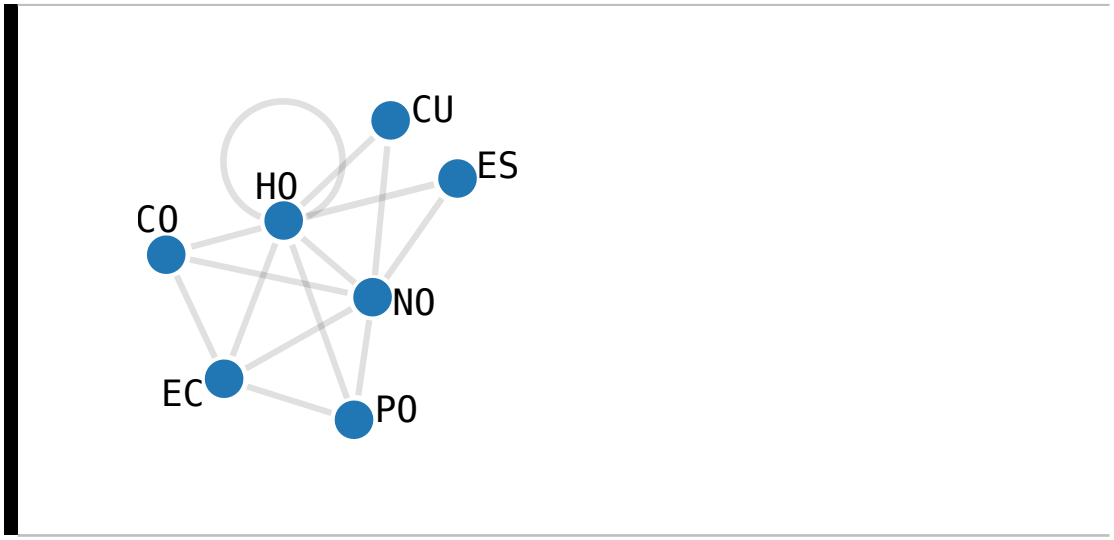
Programando de forma simplificada no SageMath, escrevemos da seguinte forma:

Código 5.4 – Ligações entre páginas do *site* no SageMath

```
1 G = Graph({'HO': ['HO','NO','EC','PO','CU','ES','CO'], 'NO': ['EC
      ','PO','CU','ES','CO'], 'EC': ['PO','CO'] })
2
3 show(G)
```

Resultado apresentado para o código acima:

■



(d) Não é possível, pois não tem uma aresta que conecta diretamente os vértices HO e CU. Uma outra forma de obter essa informação seria usando o SageMath, depois de ter definido o grafo acima, digitando **G.has_edge('PO','CU')**, que retornaria o valor **False**.

(e) É possível, pois o grafo possui uma aresta que conecta diretamente os vértices ES e NO. Usando o SageMath, poderíamos saber a mesma informação digitando **G.has_edge('ES','NO')**, que retornaria o valor **True**.

Atividade 3 [Campeonato Pernambucano - Hexagonal do Título]

UMA SOLUÇÃO:

(a) A cargo do aluno.

(b) O grafo é simples, pois não contém laços nem arestas múltiplas. Também é completo, pois cada vértice está conectado com todos os outros vértices desse grafo.

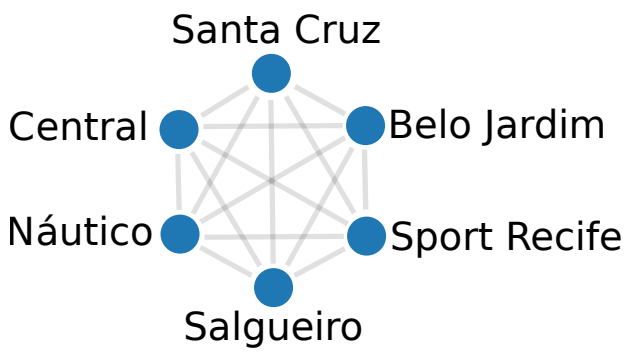
(c) Nomeando os vértices pelo nome de cada time, cada aresta conectando dois times indica a existência de dois confrontos entre essas duas equipes, sendo uma a partida de ida e outra, a de volta.

Programando no SageMath, escrevemos da seguinte forma:

Código 5.5 – Problema dos Confrontos do Pernambucano 2017 no SageMath

```
1 G = Graph({ 'Salgueiro': ['Belo Jardim','Central','Santa Cruz','
      Sport Recife','Náutico'], 'Belo Jardim': ['Central','Santa Cruz
      ','Sport Recife','Náutico'], 'Central': ['Santa Cruz','Sport
      Recife','Náutico'], 'Santa Cruz': ['Sport Recife','Náutico'], '
      Sport Recife': ['Náutico'] })
2
3 show(G)
```

Resultado apresentado para o código acima:



(d) A ordem do grafo corresponde ao número de vértices, que no caso são 6, e pode ser obtido digitando G.order() ou len(G) ou len(G.vertices()) no SageMath.

O tamanho do grafo corresponde ao número de arestas, que no caso são 15, e pode ser obtido digitando G.size() ou len(G.edges()) no SageMath.

5.2.4 Aula 4

Atividade 1 [Obtendo o grafo a partir de uma matriz de adjacência]

UMA SOLUÇÃO:

(a) Não é simples, pois a diagonal principal da matriz de adjacência apresenta um elemento igual a 1, o que indica que existe uma aresta que liga um vértice a ele mesmo, o que constitui o conceito de laço.

(b) A cargo do aluno.

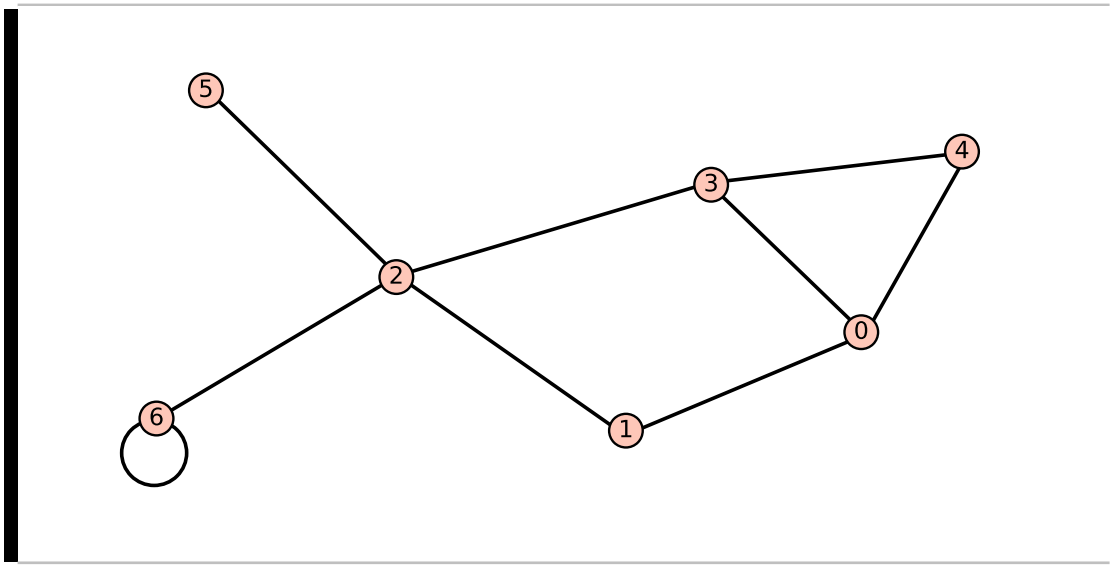
(c) A cargo do aluno.

(d) Declaramos a matriz de adjacência proposta e a convertemos em grafo, conforme apresentado no código abaixo.

Código 5.6 – Obtendo o grafo a partir da matriz de adjacência

```
1 A = Matrix([[0,1,0,1,1,0,0], [1,0,1,0,0,0,0], [0,1,0,1,0,1,1],
    [1,0,1,0,1,0,0], [1,0,0,1,0,0,0], [0,0,1,0,0,0,0],
    [0,0,1,0,0,0,1]])
2
3 G = Graph(A)
4 plot(G)
```

Resultado apresentado para o código acima:



(e) Para obter a matriz de incidência no SageMath, digitamos G.incidence_matrix(). Resultado apresentado para o código acima:

```
[1 1 1 0 0 0 0 0 0]
[1 0 0 1 0 0 0 0 0]
[0 0 0 1 1 1 1 0 0]
[0 1 0 0 1 0 0 1 0]
[0 0 1 0 0 0 0 1 0]
[0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 1 0 2]
```

(f) A cargo do aluno.

5.2.5 Aula 5

Atividade 1 [Potência 2 da Matriz de Adjacência]

UMA SOLUÇÃO:

(a) Como a matriz de adjacência A é uma matriz simétrica, então as linhas e colunas de mesmo índice apresentam as respectivas entradas iguais.

Como as entradas a_{ij} da matriz A são dadas por 1 se v_i e v_j são adjacentes e 0 no caso contrário, então cada elemento da diagonal principal de A^2 será dado pela soma dos quadrados de cada entrada da linha ou coluna que contém esse elemento.

Como cada entrada da linha ou coluna correspondente é composta por 0's e 1's, então cada elemento da diagonal principal corresponde a soma dos elementos da

linha ou coluna que o possui. Essa soma indica quantos vértices são adjacentes ao vértice correspondente à linha ou coluna e portanto, esse é exatamente o conceito de grau de um vértice.

(b) Declaramos a matriz de adjacência A do grafo de entrada, calculamos A^2, armazenamos os elementos da diagonal principal na lista de graus, tomamos o maior dentre os valores dessa lista usando o comando do Python chamado **max** e por fim, buscamos dentre os vértices desse grafo os que tem esse grau máximo, conforme apresentado no código abaixo.

Código 5.7 – Obtendo o(s) vértice(s) de maior grau a partir da diagonal principal da potência 2 de A

```
1  def maior_grau(X):
2      A = X.adjacency_matrix()
3      M = A^(2)
4
5      V = X.vertices()
6      n = len(X)
7      graus = []
8      vert = []
9
10     for i in range(n):
11         graus.append(M[i][i])
12
13     maior_grau = max(graus)
14
15     for i in range(n):
16         if graus[i] == maior_grau:
17             vert.append(V[i])
18
19     return vert, maior_grau
```

Atividade 2 [Complemento de um Grafo]

UMA SOLUÇÃO:

(a) Inicialmente fazemos uma cópia do grafo X de entrada, completamos esse grafo com as arestas que faltam, depois retiramos as arestas que estão no grafo original e por fim, imprimimos esse grafo que corresponde ao complemento do grafo X da entrada.

Código 5.8 – Obtendo o complemento de um grafo X dado

```
1  def complemento(X):
```

```
    H = copy(X)

    for i in H.vertices():
        for j in H.vertices():
            if i <> j and not (i,j) in H.edges():
                H.add_edge(i,j)

    for k in X.edges(): H.delete_edge(k)

    return plot(H)
```

(b) Como o grafo é simples, o grau máximo permitido para cada vértice é dado pelo número de vértices do grafo (v) menos uma unidade (retirando o vértice analisado).

Portanto, a sequência de graus de $\overline{G}$ em termos da sequência de graus de G é o complemento para atingir o grau máximo permitido.

(c) A cargo do aluno.

5.2.6 Aula 6

Atividade 1

UMA SOLUÇÃO:

(a) A cargo do aluno.

(b) (0-2-8-10-4-9-5) ou (0-3-8-10-4-9-5).

(c) Fazendo a leitura dos vértices e das arestas, temos:

Código 5.9 – Declarando o grafo G no SageMath

```
G = Graph({0:[2,3], 1:[8], 2:[3,8], 3:[8], 4:[7,9,10], 5:[6,9],
   6:[9], 7:[10], 8:[10]})
```

(d) Digitando G.shortest_paths(0), obtemos o resultado apresentado a seguir:

```
{0: [0], 1: [0, 2, 8, 1], 2: [0, 2], 3: [0, 3], 4: [0, 2,
8, 10, 4], 5: [0, 2, 8, 10, 4, 9, 5], 6: [0, 2, 8, 10, 4,
9, 6], 7: [0, 2, 8, 10, 7], 8: [0, 2, 8], 9: [0, 2, 8, 10,
4, 9], 10: [0, 2, 8, 10]}
```

(e) São pontes as arestas que ligam os vértices 1 e 8, 4 e 9, 8 e 10.

(f) Resultado apresentado para o comando **G.bridges()**:

```
[(1, 8, None), (4, 9, None), (8, 10, None)]
```

(g) Criamos uma cópia do grafo X, excluímos cada aresta por vez e testamos se o número de componentes conexas aumentou em relação ao grafo X, depois adicionamos de volta a aresta removida.

Código 5.10 – Função que retorna a lista de pontes de um grafo X.

```
1 def pontes(X):
2     H = X.copy()
3     L = []
4     for k in H.edges():
5         H.delete_edge(k)
6         if X.connected_components_number() < H.
      connected_components_number():
7             L.append(k)
8         H.add_edge(k)
9     return L
```

(h) Analisando os vértices do grafo G, temos como pontos de articulação os vértices 4, 8, 9 e 10.

(i) Criamos uma cópia do grafo X, excluímos cada vértice por vez e testamos se o número de componentes conexas aumentou em relação ao grafo X, depois retomamos a cópia do grafo X, pois não basta apenas readicionar o vértice deletado já que as arestas que incidiam nele não serão readicionadas.

Código 5.11 – Função que retorna a lista de vértices de corte de um grafo X.

```
1  def vertices_corte(X):
2      H = X.copy()
3      L = []
4      for k in H.vertices():
5          H.delete_vertex(k)
6          if X.connected_components_number() < H.
       connected_components_number():
7              L.append(k)
8          H = X.copy()
9
10     return L
```

Atividade 2

UMA SOLUÇÃO:

(a) Sim. Basta colocar os operários em uma partição e as máquinas em outra partição.

(b) A cargo do aluno.

(c) Não, pois há um vértice isolado gerando duas componentes e portanto, o grafo é desconexo.

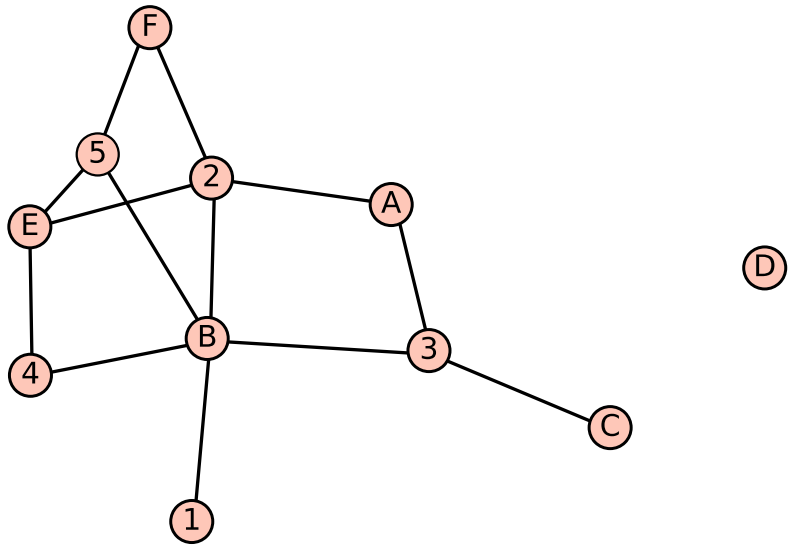
(d) Declarando o grafo no SageMath, temos:

Código 5.12 – Declarando o grafo G

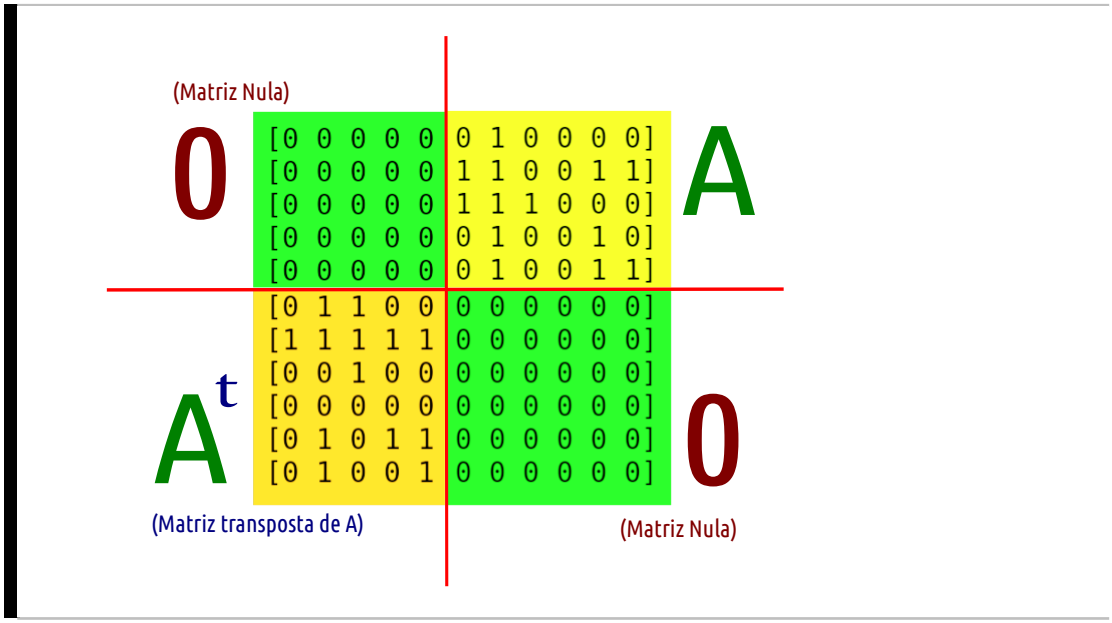
```
G = Graph({'A': [2,3], 'B': [1, 2, 3, 4, 5], 'C': [3],
  'D': [], 'E': [2, 4, 5], 'F': [2, 5]})

plot(G)
```

Resultado apresentado para o código acima:



(e) A aparência está mostrada na figura a seguir.



(f) A cargo do aluno.

(g) Se o grafo de entrada for biparido, armazenamos as bipartições em duas listas. Percorremos os elementos dessas listas e se existir aresta com extremos em cada lista dessa, atribuímos 1 na entrada correspondente da matriz e em caso contrário, atribuímos 0, conforme apresentado no código abaixo.

Código 5.13 – Função que retorna a matriz de bipartição, se existir.

```
1  def matriz_bipart(X):
2      if X.is_bipartite():
3          L = []; R = []
4          B = BipartiteGraph(X)
5          for i in B.left:
6              L.append(i)
7          for i in B.right:
8              R.append(i)
9
10         l = len(L); r = len(R); linha = []; M = []
11         print "Vértices das linhas:", L
12         print "Vértices das colunas:", R
13         print ""
14         print "Matriz da Bipartição"
15         for k in range(l):
16             linha = []
17             for t in range(r):
18                 if X.has_edge(L[k], R[t]):
19                     linha.append(1)
20                 else:
```

```
                    linha.append(0)
                M.append(linha)
            MatBip = Matrix(M)
            return MatBip
        else:
            print "Esse grafo não é bipartido!"
```

5.2.7 Aula 7

Atividade 1 [Problema do Dominó]

UMA SOLUÇÃO:

(a) Não, pois existem laços que são as carroças do dominó (peças cujas pontas tem o mesmo valor como 0-0, 1-1, 2-2, ...).

(b) A cargo do aluno.

(c) Não, pois contém laços e todo grafo completo é um grafo simples, ou seja, sem laços nem arestas múltiplas. Se removidos os laços, o grafo se torna completo, pois cada vértice está ligado a todos os outros vértices por uma aresta.

(d) Usando laços repetitivos, podemos fazer uma ponta i variar de 0 a 6 e a outra variar de i a 6, para não criar arestas iguais como em (3,5) e (5,3), conforme código abaixo.

Código 5.14 – Código para obter as arestas do grafo do dominó.

```
G = Graph(loops=True)
for i in range(7):
    for j in range(i,7):
        G.add_edge(i,j)
```

(e) A pergunta é correspondente a saber se G é euleriano. Como cada vértice do grafo G contém um laço e está ligado aos outros 6 vértices, então todo vértice de G tem grau 8, que é par.

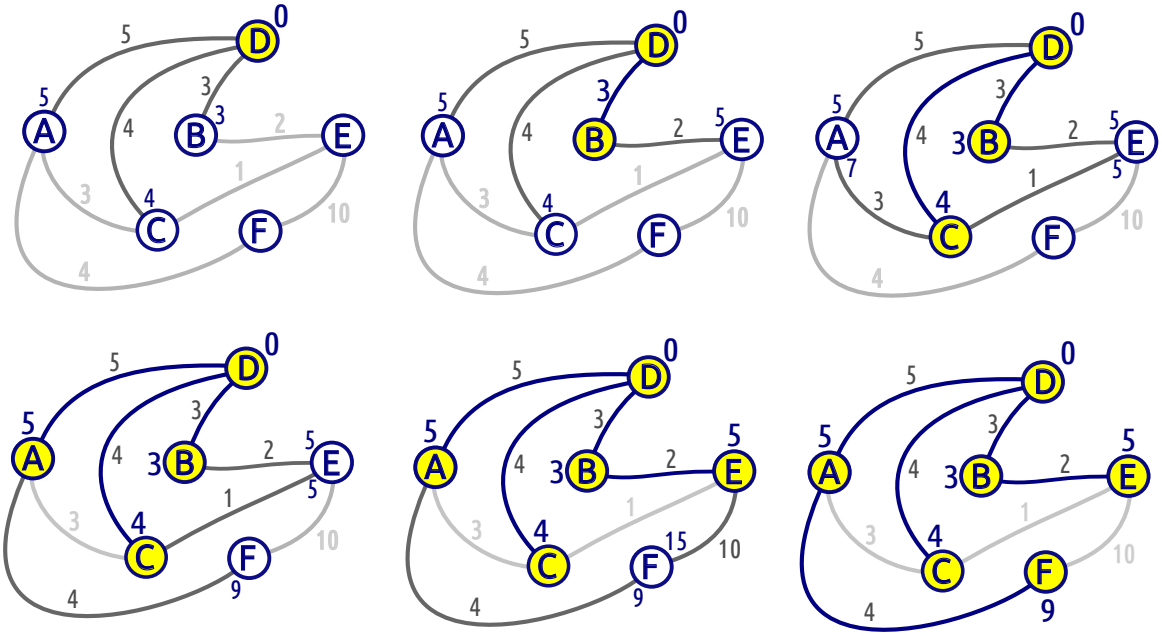
Logo, pelo Teorema de Euler, G é euleriano e portanto, é possível com as pedras de um jogo de dominó formar um anel de acordo com a regra do dominó como apresentado na figura.

(f) Como cada vértice do grafo G correspondente a esse novo dominó contém um laço e está ligado aos outros 7 vértices, então todo vértice desse grafo G teria grau 9, que é ímpar, e portanto, pelo Teorema de Euler, G não seria euleriano, isto é, seria impossível com as pedras desse novo jogo de dominó formar um anel de acordo com a regra do dominó como apresentado na figura.

Atividade 2

UMA SOLUÇÃO:

Figura 46 – Solução do problema do menor caminho.



5.2.8 Aula 8

Atividade 1

UMA SOLUÇÃO:

(a) Sim, pois esse grafo não contém ciclos e é conexo.

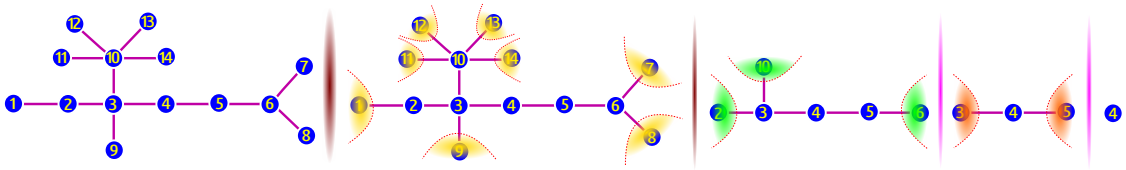
(b) As folhas são 1, 7, 8, 9, 11, 12, 13 e 14, pois todos esses vértices tem grau 1.

(c) Não é possível a comunicação direta entre as pessoas 6 e 11, pois não existe aresta que os conectam.

A sequência em que a pessoa 13 pode se comunicar com a passoa 8 é única e se trata do caminho que liga os vértices 8 e 13, a saber: 13 - 10 - 3 - 4 - 5 - 6 - 8.

Conforme o Teorema 18, em uma árvore, quaisquer dois vértices são conectados por um único caminho, e portanto, como o grafo apresentado é uma árvore, não existe mais de uma forma de comunicação entre duas pessoas quaisquer.

(d) Como se trata de uma árvore, podemos eliminar as folhas a cada passo até sobrar um ou dois vertices centrais. A solução segue abaixo.



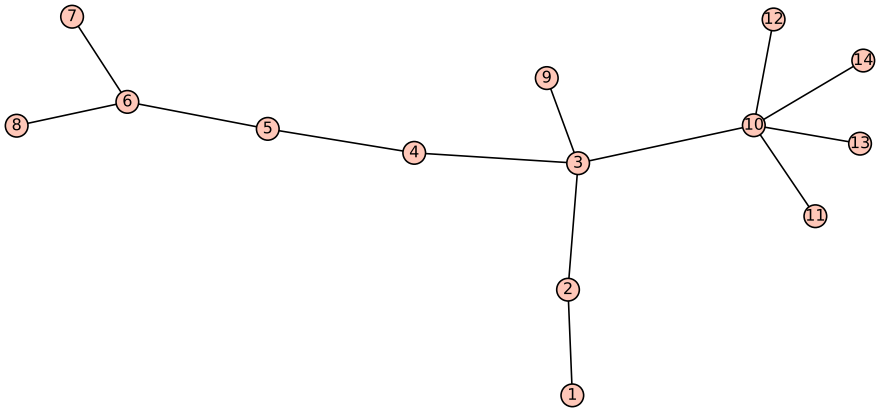
Portanto, o centro da árvore é o vértice 4, ou seja, num contexto aplicado, o líder do grupo que facilitaria o acesso às pessoas é a pessoa 4.

(e) Declarando o grafo no SageMath e visualizando esse grafo, podemos escrever conforme o código abaixo.

Código 5.15 – Declarando os grafos G e H no SageMath

```
1  G = Graph({1:[2], 3:[2,4,9,10], 4:[5], 6:[5,7,8],
       10:[11,12,13,14]})
2
3  plot(G)
```

Resultado apresentado para o código acima:



(f) O resultado obtido ao executar o comando **G.center()** no SageMath está apresentado abaixo.

```
[4]
```

(g) Para fazer a analise se X é uma árvore, usamos X.is_tree(), em caso positivo, imprimimos que o grafo é uma árvore; em caso contrário, que não é uma árvore.

Para retornar o centro desse grafo a partir da matriz de distância, tomamos a matriz de distância (X.distance_ matrix()) e percorremos cada linha buscando o maior valor, que é a excentricidade de cada vértice, e acrescentamos em um lista L.

Tomamos o mínimo dessa lista L e percorrendo essa lista com a excentricidade de cada vértice, buscando os vértices cuja excentricidade coincide como o mínimo obtido e acrescentamos na lista R que contém o centro do grafo X.

A ideia acima está escrito a seguir.

Código 5.16 – Função que retorna o centro do grafo X e a informação se X é uma árvore.

```
def centro(X):
    if X.is_tree():
        print "O grafo é uma árvore!"
    else:
        print "O grafo não é uma árvore!"

    M = X.distance_matrix()
    n = len(X)
    L = []; R = []; V = X.vertices()

    for i in range(n):
        L.append(max(M[i]))

    t = min(L)
    for k in range(n):
        if L[k] == t:
            R.append(V[k])

    return R
```

Atividade 2

UMA SOLUÇÃO:

(a) O grafo H, pois não há ciclos e o grafo é conexo.

(b) A cargo do aluno.

(c) Como H é uma árvore, podemos usar a maneira de determinar o centro de uma árvore apresentado no item (d) da atividade 1. Eliminando as folhas a cada passo, temos que o centro do grafo H é formado pelos vértices A e B.

Como o grafo G não se trata de uma árvore, precisamos calcular a excentricidade de cada vértice: E(1) = 4, E(2) = 4, E(3) = 3, E(4) = 3, E(5) = 3, E(6) = 3, E(7) = 3, E(8) = 3, E(9) = 4 e E(10) = 4. Como o centro é formado pelos vértices com menor excentricidade, temos que o centro é formado pelos vértices 3, 4, 5, 6, 7 e 8.

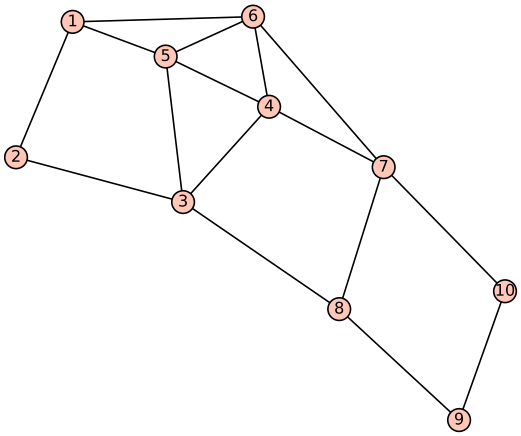
(d) Declarando os grafos no SageMath e visualizando-os, podemos escrever conforme o código abaixo.

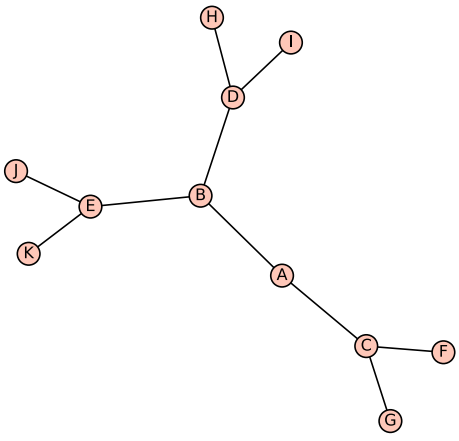
Código 5.17 – Declarando os grafos G e H no SageMath

```
G = Graph({1:[2,5,6], 3:[2,4,5,8], 4:[5,6], 6:[5,7], 7:[4,8,10],
    9:[8,10]})
H = Graph({'A':['B','C'], 'B':['D','E'], 'C':['F','G'], 'D':['H','I
    '], 'E':['J','K']})

plot(G)
plot(H)
```

Resultado apresentado para o código acima:



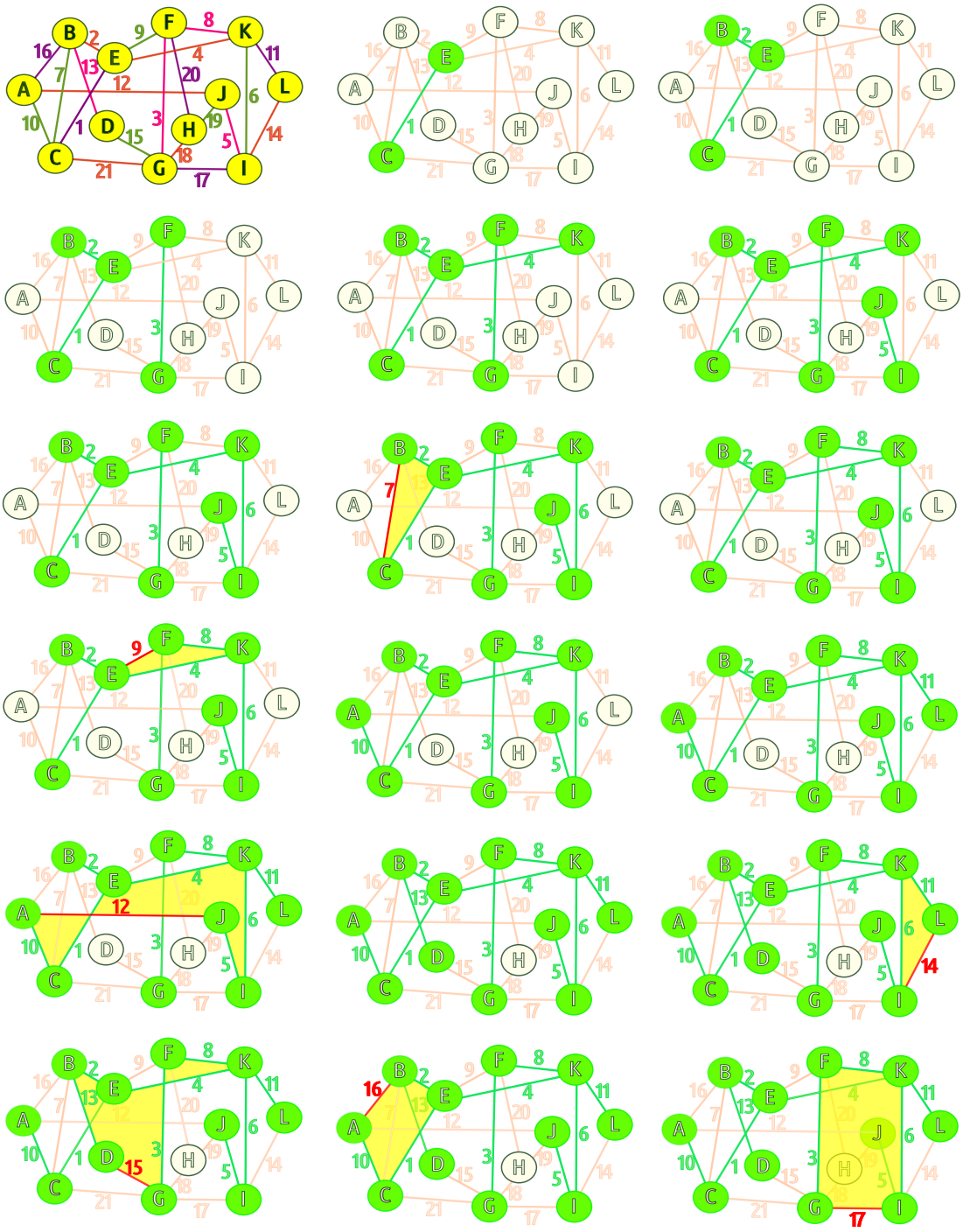


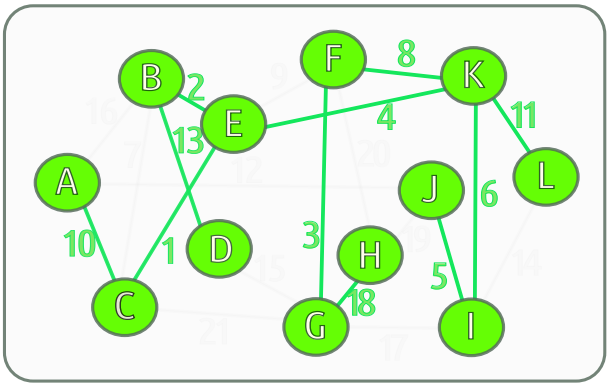
(e) A cargo do aluno.

Atividade 3 [Árvore Geradora]

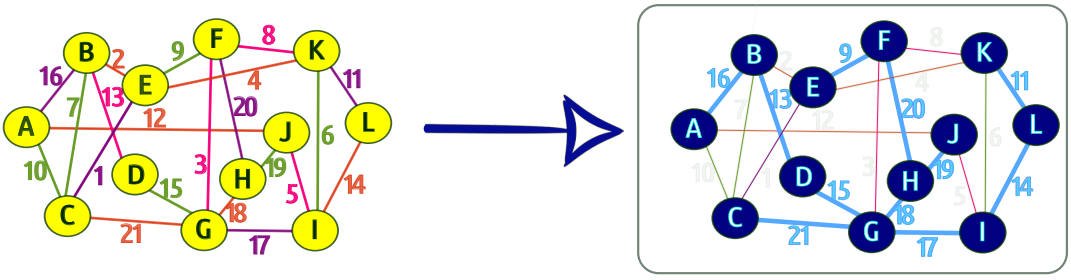
UMA SOLUÇÃO:

(a) Aqui apresentaremos apenas a solução usando o algoritmo de Kruskal para obter uma árvore geradora mínima, sendo a mesma solução se obtida pelo algoritmo de Prim.





A árvore geradora máxima pode ser obtida adaptando o algoritmo de Kruskal ou de Prim, selecionando as arestas de maior valor, sem que formem ciclos. Aqui apresentaremos o resultado final:



(b) Os pesos das árvores geradoras mínima e máxima são, respectivamente, 81 e 173.

Considerações Finais

Esse presente trabalho tem como objetivo contribuir com um material teórico sobre a Teoria dos Grafos, aliado ao poder da Ciência da Computação, que possa ser aproveitado tanto por alunos da graduação do curso de Licenciatura em Matemática quanto por professores de Matemática do Ensino Médio.

Em um mundo cada vez mais tecnológico, acreditamos que as novas tecnologias tem tornado as pessoas mais dependentes da máquina até para situações comuns do cotidiano. Em muitos casos, essa dependência vem fazendo com que as pessoas diminuam os estímulos de pensamentos e ideias, deixando de exercitar a criatividade.

Acreditamos fortemente que o grande truque de aprender algo não é observar o resultado final, mas sim entender o porquê das coisas e discutir as possibilidades que surgem no decorrer da solução dos problemas, ou seja, não é a chegada que importa, mas sim como enfrentamos os percalços da caminhada até a chegada.

Tornar o aprendizado da matemática mais prazeroso requer muito mais do que imaginamos, é preciso repensar a forma com que ensinamos e aprendemos, e mergulhar no mesmo universo em que os alunos vivem atualmente pode instigar neles o desejo pela matemática e pelo racícionio lógico. Antes de sermos reféns das máquinas e da tecnologia, precisamos perceber a maior máquina que temos, o nosso cérebro, que diferente das outras tem a capacidade de enxergar as possibilidades nos desafios que enfrentamos.

Nesse trabalho, abordamos a Teoria dos Grafos como uma possibilidade a ser inserida no currículo do Ensino Médio, aliando os fundamentos teóricos dessa área da matemática ao poder da programação, fazendo o uso do *software* SageMath, permitindo ao aluno o conhecimento de outras ferramentas que trazem vantagens na agilidade de obtenção da solução de um problema.

A partir da teoria apresentada, buscamos trabalhar as temáticas abordadas fazendo uso de exercícios mais contextualizados que permitissem aos alunos perceber a relação entre a Teoria dos Grafos e situações do cotidiano.

Deixamos aqui como primeira proposta que professores e alunos de graduação explorem outras possibilidades de conteúdos para o Currículo, indo além dos tradicionais, e como segunda proposta, que façam uso da programação em outros tópicos de Matemática, explorando a teoria com o uso da tecnologia e permitindo que os alunos possam ter outra visão dos conteúdos abordados em sala de aula.

Referências Bibliográficas

[1] BRASIL. Ministério da Educação. Secretaria da Educação Básica. **Orientações Curriculares para o Ensino Médio. Volume 2. Ciências da Natureza, Matemática e suas Tecnologias**. Brasília: MEC, 2006. 135 p. Disponível em: <http://portal.mec.gov.br/seb/arquivos/pdf/book_volume_02_internet.pdf>. Acesso em: 30 Jul 2018.

[2] BRASIL. **Parâmetros Curriculares Nacionais: Ensino Médio**. Brasília: MEC. Vol. 3, 1998. 58 p. Disponível em: <http://portal.mec.gov.br/seb/arquivos/pdf/ciencian.pdf> Acesso em: 30 Jul 2018.

[3] FONSECA FILHO, Cleúzio. **História da Computação: O caminho do pensamento e da tecnologia**. Porto Alegre : EDIPUCRS, 2007. 205 p. Disponível em: <http://www.pucrs.br/edipucrs/online/historiadacomputacao.pdf>. Acesso em: 16 Nov 2018.

[4] CODE.ORG. **What Most Schools Don't Teach**. 26 Fev 2013. (5min43s). Disponível em: <https://youtu.be/nKIu9yen5nc>. Acesso em: 20 Mar 2018.

[5] TODOS PELA EDUCAÇÃO. **De Olho nas Metas 2015-16**. 2017. Disponível em: <https://www.todospelaeducacao.org.br//arquivos/biblioteca/olho_metas_2015_16_final.pdf> Acesso em: 20 Mar 2018.

[6] KAFAI, Y.B.; BURKE, Q. **Computer Programming Goes Back to School**. 2013. In: Education Week. Disponível em: <https://www.edweek.org/ew/articles/2013/09/01/kappan_kafai.html>. Acesso em: 21 Mar 2018.

[7] ARAÚJO, P.O.; SILVEIRA, E.C.; RIBEIRO, A.M.V.B.; SILVA, J.D. **Promoção da saúde do idoso: a importância do treino da memória**. Revista Kairós Gerontologia, 15(8), pp. 169-183. Online ISSN 2176-901X. Print ISSN 1516-2567. São Paulo (SP), Brasil: FACHS/NEPE/PEPGG/PUC-SP, Dez 2012.

[8] GERALDES, Wendell Bento. **Programar é bom para as crianças? Uma visão crítica sobre o endino de programação nas escolas**. Texto Livre: Linguagem e Tecnologia, v. 7, n. 2, pp. 105-117, 2014.

[9] RAPKIEWICZ, E. C. **Estratégias pedagógicas no ensino de algoritmos e programação associada ao uso de jogos educacionais**. Revista RENOTE - Revista Novas Tecnologias na Educação, v.4, n.2, 2006.

[10] PONTE, J. P. **O computador na educação Matemática (Cadernos de Educação Matemática, Nº 2)**. Lisboa: Editora APM, 1991.

[11] BALESTRI, Rodrigo Dias. **A participação da História da Matemática na Formação de Professores de Matemática na Óptica de Professores/Pesquisadores**. 106 f. Dissertação (Mestrado em ensino de ciências e educação matemática) - Universidade Estadual de Londrina, Paraná, 2008.

[12] PYTHON.ORG. **Python 3.5.5 documentation**. Disponível em: <https://docs.python.org/3.5/index.html>. Acesso em: 04 Jun 2018.

[13] GOLDBERG, M.; GOLDBERG, E. **Grafos. Conceitos, Algoritmos e Aplicações** . 1ª Edição (2012). Brasil: Elsevier Editora Ltda, 2012.

[14] JURKIEWICZ, Samuel. **Grafos: Uma Introdução**. Apostila 5 - Programa de Iniciação Científica (PIC) - OBMEP. 2007. Disponível em: <http://www.obmep.org.br/docs/apostila5.pdf>. Acesso em: 04 Jun 2018.

[15] BONDY, J. A.; MURTY, U. S. R. **Graph Theory with Applications**. Canadá: Editora Elsevier Science Ltd/North-Holland, 1976.

[16] DIESTEL, Reinhard. **Graph Theory**. Graduate Texts in Mathematics, Volume 173. 5ª Edição (2016). Heidelberg: Editora Springer-Verlag, 2005.

[17] BONDY, J. A.; MURTY, U. S. R. **Graph Theory**. Graduate Texts in Mathematics, Volume 244. New York: Editora Springer-Verlag, 2008.

[18] FEOFILOFF, Paulo. **Algoritmos para Grafos em C via Sedgewick**. Disponível em: <https://www.ime.usp.br/ pf/algoritmos_para_grafos/index.html>. Acesso em: 04 Jun 2018.

[19] ALVES, Robson Piacente. **Coloração de grafos e aplicações**. 132 f. Dissertação (Mestrado Profissional em Matemática) - Universidade Federal de Santa Catarina, Florianópolis, 2015. p. 106-111.

[20] SAGEMATH.ORG. **Sage Reference Manual: Graph Theory**. Disponível em: <https://doc.sagemath.org/pdf/en/reference/graphs/graphs.pdf>. Acesso em: 04 Jun 2018.