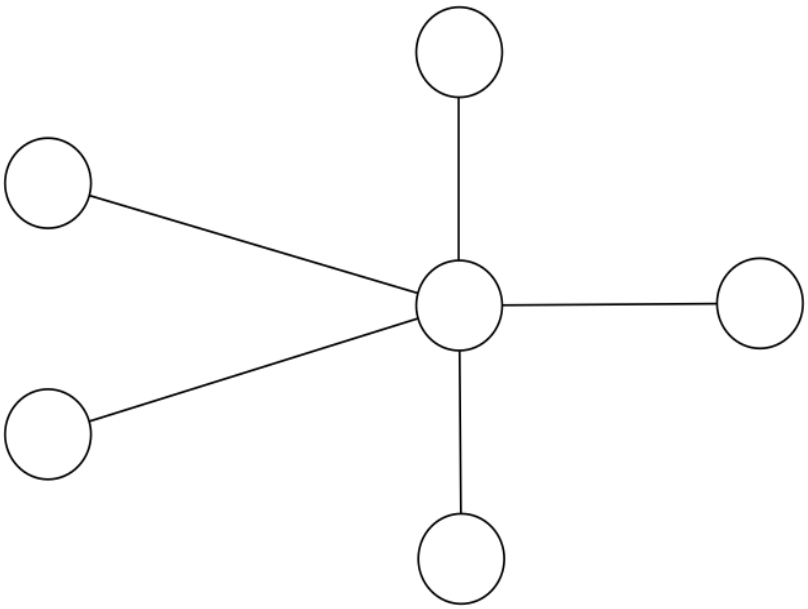
Algoritmos em Grafos

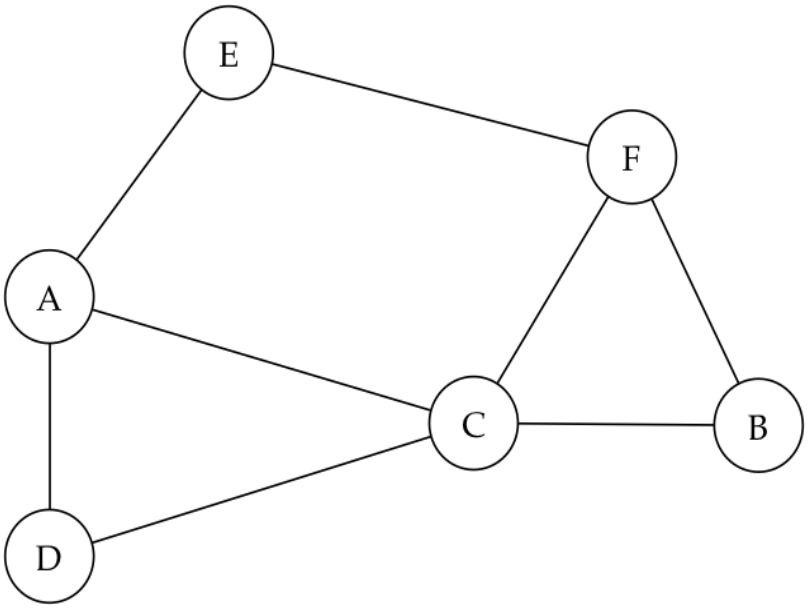
Grafos são importantíssimos em Ciência da Computação. Eles aparecem, por exemplo, na modelagem de problemas envolvendo distâncias entre pontos em um mapa, na modelagem de redes sociais, de ligações (links) entre páginas na Internet, e em inúmeros outros cenários.

Dada a importância de grafos em Ciência da Computação, saber modelar problemas usando grafos e dominar os algoritmos que operam sobre grafos são habilidades extremamente úteis para um programador. Feita esta propaganda inicial, podemos partir de fato para nosso estudo de algoritmos em grafos.

O nome *grafo* pode parecer estranho, mas grafos são, na verdade, coisas relativamente simples. A primeira coisa a se saber é que um *grafo* é diferente de um *gráfico*, aquela coisa com o eixo xx e o eixo yy que a gente estuda em matemática. Um grafo, por sua vez, nada mais é do que um conjunto de vértices (que em nos nossos desenhos de grafos serão representados como bolinhas) conectados por arestas (que em nos nossos desenhos serão representadas por linhas ou setas). Assim, um grafo representa relacionamentos (expressados pelas arestas) entre um conjunto de entidades (os vértices).

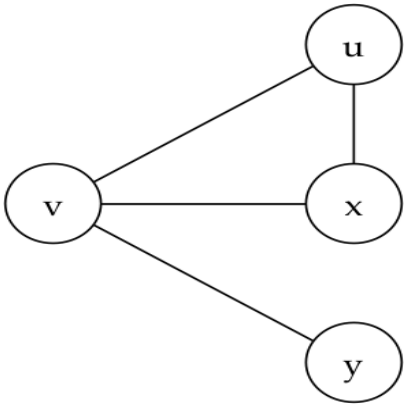


O interessante de grafos é que com algo aparentemente simples conseguimos modelar uma multitude de problemas interessantes. Por exemplo, suponha que desejemos representar as ruas e esquinas de uma cidade como um grafo. Para isso, basta considerarmos as esquinas como vértices e as ruas como arestas. Com essa modelagem simples, podemos, por exemplo, determinar, dadas as esquinas A e B, qual é o número mínimo de esquinas pelas quais precisamos passar para ir de A até B. Um exemplo de grafo como o que acabamos de descrever é mostrado abaixo.

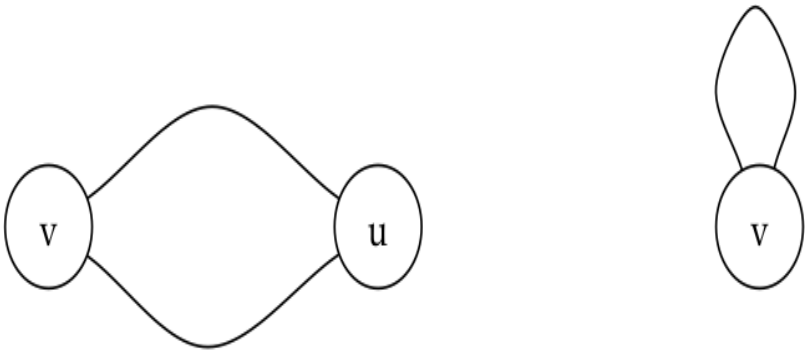


Definições

Como dissemos anteriormente, um grafo consiste de um conjunto de vértices (ou nodos) com arestas conectando alguns deles. Usaremos VV para nos referir ao conjunto de vértices de um grafo e EE para nos referir ao conjunto de arestas. Se existe uma aresta entre dois vértices, dizemos que esses vértices são **vizinhos**. O **grau** de um vértice é o número de vizinhos que ele possui. Na figura abaixo, os seguintes pares de vértices são vizinhos: (v, u), (v, x), (v, y), e (u, x).



A menos que especifiquemos o contrário, os grafos com os quais lidaremos não possuem múltiplas arestas entre os mesmos vértices (multi-arestas) nem arestas saindo de um vértice e chegando nele mesmo (self-loops). A figura abaixo ilustra esses cenários.

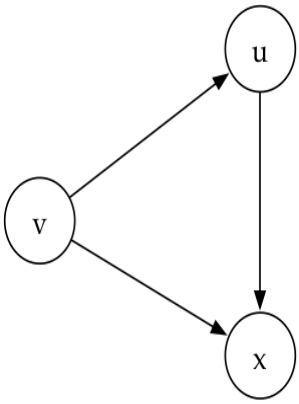


a) b)

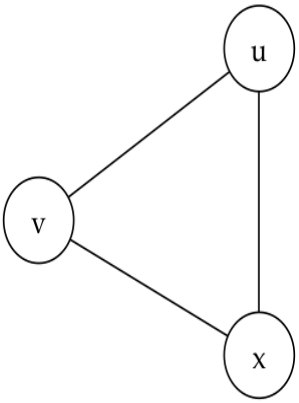
Em geral, representamos o número de vértices de um grafo usando a letra nn e o número de arestas usando a letra mm. De modo mais formal, temos n=|V|n=|V| (leia-se nn é igual à cardinalidade, ou tamanho, do conjunto de vértices) e m=|E|m=|E|.

No grafo descrito acima (com vértices representando esquinas e arestas representando ruas), as conexões entre vértices são bidirecionais, ou seja, é possível usar uma rua tanto para ir quanto para vir de uma determinada esquina (as ruas são de

mão dupla). Entretanto, é possível ter grafos em que as conexões entre vértices são unidirecionais. Tais grafos são chamados **grafos direcionados**, enquanto os outros (com conexões de mão dupla) são denominados **grafos não-direcionados**.



a)



b)

Até o momento, vimos como representar grafos de forma visual.

Representação de Grafos

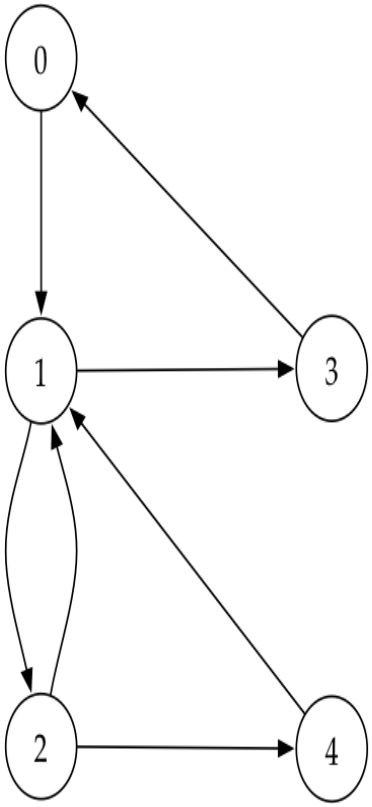
Na seção anterior, vimos os conceitos principais sobre grafos e aprendemos como eles podem ser visualizados. Nesta seção veremos como eles podem ser representados em um programa de computador.

A melhor representação para um grafo depende do que desejamos fazer com ele e do quão denso ou esparso ele é. Grafos esparsos são grafos que possuem um pequeno número de arestas em relação ao número de vértices, ao passo que em grafos densos o número de arestas se aproxima do máximo de arestas possível.

Existem duas representações principais para grafos. A primeira é chamada **matriz de adjacências**, que nada mais é do que uma matriz MM de nn linhas e nn colunas em que M[i,j]=1M[i,j]=1 se existe uma aresta do vértice ii para o vértice jj. Lembre-se que estamos seguindo a convenção de que nn representa o número de vértices do grafo.

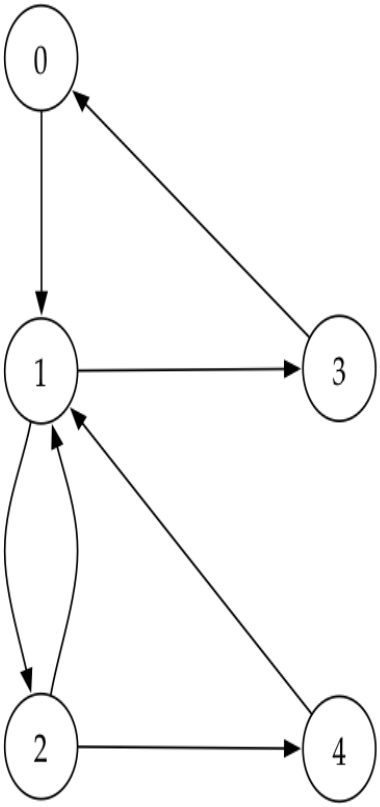
Se estivermos representando um grafo não-direcionado e existir uma aresta entre os vértices ii e jj, então teremos tanto M[i,j]=1M[i,j]=1 quanto M[j,i]=1M[j,i]=1. Em outras palavras, a matriz de adjacências MM que representa um grafo não-direcionado é simétrica. Nessa representação, o tempo gasto para determinar se um vértice uu é vizinho de um vértice vv é constante, pois basta verificarmos se M[u,v]M[u,v] é igual a 11.

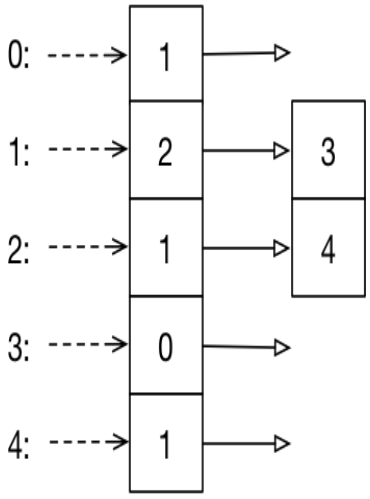
Vejamos um exemplo de grafo e sua matriz de adjacências.



	0	1	2	3	4
0		1			
1			1	1	
2		1			1
3	1				
4		1			

A segunda representação é chamada **lista de adjacências**, e consiste de um arranjo (ou lista) AA com nn nodos em que A[i]A[i] contém a lista de vizinhos do vértice ii. Se o grafo for direcionado, os vizinhos de um vértice vv serão somente aqueles vértices para os quais existe uma aresta saindo de vv. Nessa representação, o tempo gasto para determinar se um vértice uu é vizinho de um vértice vv é proporcional ao grau de saída do vértice vv, ou seja, ao número de arestas que saem de vv.





Se o grafo a ser representado for relativamente esparso, a lista de adjacências oferecerá uma representação mais compacta (econômica) do grafo porque ela conterá somente as conexões existentes entre vértices.

Representando Grafos em Python

Como listas de adjacência são a forma mais comum de se representar grafos, nosso foco nesta seção será em como implementar um grafo em Python usando essa estrutura de dados.

Nesta seção veremos duas formas simples de representar listas de adjacência em Python. Uma delas, que chamamos de *representação explícita*, armazena os nomes dos vértices no próprio grafo. A outra, chamada *representação implícita*, assume que cada vértice possui um identificador inteiro, e faz referência aos vértices por meio desse identificador.

Antes de discutirmos as formas de representar grafos, convém esclarecermos como montamos o grafo em si. Em nossos exemplos, assumiremos que nos será fornecida uma lista com as

ligações entre vértices (uma lista de arestas), a partir da qual criaremos os vértices e as arestas que os conectam. Nos exemplos abaixo, trabalharemos com a seguinte lista de arestas:

```
A B
B C
B D
C B
C E
D A
E B
```

Na lista acima, existe uma aresta saindo do vértice à esquerda e chegando no vértice à direita. Por exemplo, a aresta `A B` significa que existe uma aresta saindo do vértice `A` e chegando no vértice `B`.

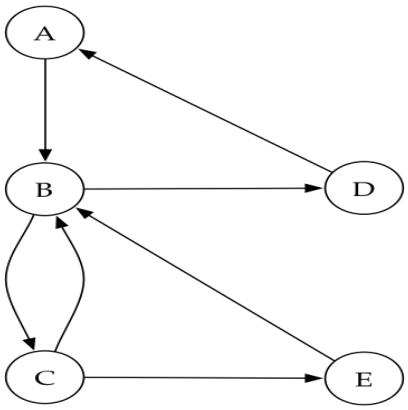
Vejamos como representar o grafo descrito pela lista de arestas acima.

Representação Explícita

De uma forma bem simples, podemos representar um grafo em Python assim:

```
grafo = { "A" : ["B"],
          "B" : ["C", "D"],
          "C" : ["B", "E"],
          "D" : ["A"],
          "E" : ["B"]
        }
```

O código acima representa o grafo como um dicionário de listas. Ele nos diz que existe uma aresta do vértice `A` para o vértice `B`, uma aresta do vértice `B` para os vértices `C` e `D`, e assim por diante. A figura abaixo mostra o grafo representado pelo código acima.



O interessante de se usar um dicionário para representar o grafo é a conveniência que isso proporciona. As chaves do dicionário são os vértices e os valores são as listas de adjacências. Para iterar sobre cada vértice `v` de um grafo `G`, precisamos simplesmente fazer `for v in G`. E para iterar sobre os vizinhos de um vértice `v`, basta fazer `for u in G[v]`.

Outra alternativa é representar o grafo como um dicionário de conjuntos. Na prática, muito pouca coisa muda em relação à representação acima, mas se usarmos um conjunto em vez de uma lista garantimos que não existirão arestas duplicadas entre dois nodos – algo que assumimos para todos os grafos com que trabalharemos nos exemplos.

Antes de avançarmos, é **muito importante** mencionar que se você precisa criar e manipular um grafo em Python de forma simples e rápida, a forma mostrada acima é provavelmente sua melhor opção. A maioria dos algoritmos em grafos pode ser implementada usando essa estrutura como base. Ela também é provavelmente a melhor opção caso você não precise fazer nada muito sofisticado (como ter programas externos acessando sua implementação de grafo, por exemplo). Nas seções seguintes, trabalharemos com essa implementação simples de grafo.

Porém, se você deseja algo mais elaborado, você pode criar uma classe para representar seu grafo, como explicamos a seguir.

Criando uma Classe para Representar Grafos em Python

Dado um grafo qualquer, precisamos realizar operações sobre ele. As operações mais comuns são obter a lista de vértices do grafo, obter a lista de arestas, verificar se existe uma aresta entre dois vértices, adicionar uma aresta entre dois vértices, etc. No exemplo abaixo, criamos uma classe `Grafo` na qual implementamos essas e outras operações.

```
from collections import defaultdict


class Grafo(object):
    """ Implementação básica de um grafo. """

    def __init__(self, arestas, direcionado=False):
        """Inicializa as estruturas base do grafo."""
        self.adj = defaultdict(set)
        self.direcionado = direcionado
        self.adiciona_arestas(arestas)


    def get_vertices(self):
        """ Retorna a lista de vértices do grafo. """
        return list(self.adj.keys())


    def get_arestas(self):
        """ Retorna a lista de arestas do grafo. """
        return [(k, v) for k in self.adj.keys() for v in self.adj[k]]


    def adiciona_arestas(self, arestas):
        """ Adiciona arestas ao grafo. """
        for u, v in arestas:
            self.adiciona_arco(u, v)


    def adiciona_arco(self, u, v):
        """ Adiciona uma ligação (arco) entre os nodos 'u' e 'v'. """
        self.adj[u].add(v)
        # Se o grafo é não-direcionado, precisamos adicionar arcos nos dois
sentidos.
        if not self.direcionado:
            self.adj[v].add(u)


    def existe_aresta(self, u, v):
        """ Existe uma aresta entre os vértices 'u' e 'v'? """
        return u in self.adj and v in self.adj[u]


    def __len__(self):
        return len(self.adj)
```

```
    def __str__(self):
        return '{}({})'.format(self.__class__.__name__, dict(self.adj))


    def __getitem__(self, v):
        return self.adj[v]
```

Alguns comentários sobre a implementação acima:

- Ela nos permite criar tanto grafos direcionados quanto não direcionados, por meio do parâmetro `direcionado` usado ao criar o grafo.
- Ela assume que todas as arestas possuem peso 1, ou seja, o grafo é não-ponderado. Em breve mostraremos como implementar grafos ponderados.
- Implementamos dois métodos mágicos: *`len`* e *`str`*. Estes métodos são implementados para fazer com que o acesso à nossa estrutura de dados seja feito como o acesso a estruturas de dados padrão do Python. Por exemplo, para que sejamos capazes de obter o tamanho do grafo usando `len(grafo)`, precisamos implementar o método *`len`*. Caso contrário, precisaríamos fazer `len(grafo.adj)`, mas isso é ruim porque ao fazer isso estamos acessando a implementação do grafo em si. Se essa implementação mudar no futuro (se o nome `adj` mudar, por exemplo) nosso código não funcionará mais. Para prevenis quanto a acessos à implementação do grafo, precisamos prover métodos acessórios, como fizemos. Para que possamos fazer `print(grafo)` e obter algo legível, precisamos implementar o método *`str`*. E para que ao fazer `grafo[v]` obtenhamos a lista de adjacências do vértice `v`, precisamos implementar o método *`getitem`*. Veja mais detalhes sobre esse tipo de método aqui.

Vejamos como criar e imprimir um grafo.

```
# Cria a lista de arestas.
arestas = [('A', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'B'), ('C', 'E'), ('D',
'A'), ('E', 'B')]

# Cria e imprime o grafo.
grafo = Grafo(arestas, direcionado=True)
```

```
print(grafo.adj)
defaultdict(<class 'set'>, {'A': {'B'}, 'B': {'C', 'D'}, 'C':
{'B', 'E'}, 'D': {'A'}, 'E': {'B'}})
```

Vejamos agora como usar algumas das funções da classe `Grafo` implementada acima.

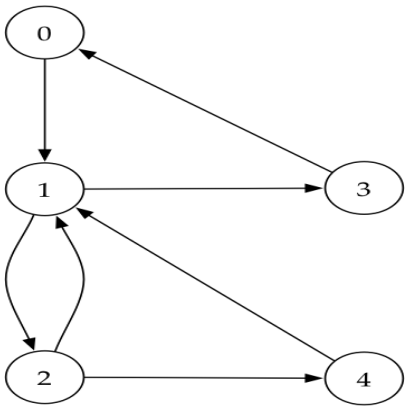
```
print(grafo.get_vertices())
['A', 'B', 'C', 'D', 'E']
print(grafo.get_arestas())
[('A', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'B'), ('C', 'E'),
('D', 'A'), ('E', 'B')]
print(grafo.existe_aresta('A', 'B'), grafo.existe_aresta('E', 'C'))
True False
```

Representação Implícita

A diferença entre a representação explícita e a implícita é que nesta cada vértice recebe automaticamente um identificador inteiro enquanto naquela o nome de um vértice é definido arbitrariamente. Vejamos um exemplo de grafo representado implicitamente.

```
grafo = [ [1],          # Vizinhos do vértice 0.
          [2, 3],       # Vizinhos do vértice 1.
          [1, 4],       # Vizinhos do vértice 2.
          [0],          # Vizinhos do vértice 3.
          [1]           # Vizinhos do vértice 4.
        ]
```

O grafo acima mostra uma representação equivalente à do grafo com o qual vínhamos trabalhando até agora (o vértice 0 é equivalente ao A, o vértice 1 ao B, e assim por diante). Veja abaixo uma figura ilustrando esse grafo.



Como exercício, veja se consegue adaptar a implementação da classe `Grafo` para essa nova representação. Aqui está uma sugestão de implementação. Basicamente, a única coisa que muda é a declaração da lista de adjacências. Antes, ela era declarada assim:

```
self.adj = defaultdict(set)
```

Agora, essa mesma estrutura pode ser declarada assim:

```
self.adj = [[] for _ in range(num_vertices)]
```

Note que, como estamos representando cada vértice do grafo por um identificador inteiro (a posição do vértice na estrutura `adj`), precisamos saber de antemão o número de vértices do grafo para inicializar a lista de adjacências. Na prática, a implementação usando dicionário é mais prática, mas pode ser mais lenta em alguns casos, pois, apesar complexidade de se acessar um elemento em um dicionário ser constante (como ocorre com as listas em Python), no pior caso essa complexidade é linear no tamanho do dicionário.

Ordenação Topológica

Ordenação topológica é uma das principais aplicações de busca em profundidade, portanto merece nossa atenção. Primeiro, explicaremos (por meio de exemplos) o que é o problema de ordenação topológica. Depois, definiremos mais formalmente o problema, e por fim mostraremos dois algoritmos para resolvê-lo.

O professor Donald Knuth apresenta um exemplo interessante de ordenação topológica. Nesse exemplo, imagine um grande glossário contendo definições de termos técnicos. O problema de ordenação topológica neste caso é encontrar uma forma de organizar as palavras no glossário de modo que nenhum termo seja usado antes de ser definido.

Pelo exemplo acima, vemos que ordenação topológica está relacionada com *dependências* entre entidades que desejamos representar: termos do glossário dependem de outros termos que tenham sido definidos anteriormente.

Seguindo essa mesma ideia de dependência, vejamos outro exemplo. Suponha que lhe seja fornecida uma lista de tarefas, na qual cada tarefa possui uma lista de outras tarefas que devem ser completadas antes dela. Seu objetivo é criar uma lista na qual as tarefas devem ser executadas obdecendo as dependências entre tarefas. Esse problema pode ser modelado como um grafo cujos vértices são as tarefas e cujas arestas são dependências entre tarefas. A solução do problema é a ordenação topológica do grafo.

Em outras palavras, a ordenação topológica é uma forma de listar os vértices de um grafo de modo que para cada aresta (u,v) (ou seja, saindo de u e chegando em v) do grafo, u aparece antes de v na lista. Um requisito básico da ordenação topológica é que o **grafo não pode ter ciclos** (uma tarefa A que depende de B que depende de A, por exemplo).

Ainda outro exemplo de ordenação topológica, encontrado no livro "Introduction to Algorithms", escrito por Cormem e outros, é o de como se vestir. Segundo esse exemplo, para nos vestirmos (corretamente) é preciso obedecer uma ordem entre peças de vestuário. Por exemplo, precisamos colocar as meias antes de colocar os sapatos, precisamos colocar a roupa de baixo (cueca ou calcinha) antes de vestir as calças, e assim por diante. Esse problema pode ser modelado como o problema de se encontrar a ordenação topológica do grafo cujos vértices são peças do vestuário e cujas arestas são dependências entre essas peças.

Analisando o exemplo anterior, é possível concluir que o Super-Homem não sabe fazer ordenação topológica, dado que ele veste a cueca por cima da calça!

Piadas à parte, vamos agora definir o problema mais precisamente e analisar o funcionamento de dois algoritmos de ordenação topológica.

O problema de ordenação topológica pode ser definido assim:

Definição: Dado um grafo acíclico direcionado $G(V,E)$ com n vértices, atribua números de 0 a $n-1$ aos vértices de modo que, se v receber o número k então todos os vértices que podem ser alcançados a partir de v recebem números maiores que k.

Implementação Baseada em Busca em Profundidade

Para os algoritmos abaixo, por questões de simplicidade, usaremos a representação do grafo como lista de listas (representação implícita), como mostrado abaixo:

```
grafo = [ [1],          # Vizinhos do vértice 0.
          [2, 3],       # Vizinhos do vértice 1.
          [1, 4],       # Vizinhos do vértice 2.
          [0],          # Vizinhos do vértice 3.
          [1]           # Vizinhos do vértice 4.
        ]
```

O algoritmo de ordenação topológica consiste de três passos principais:

1. Execute o algoritmo de busca em profundidade no grafo e mantenha registro dos tempos em que cada vértice terminou de ser processado.
2. No momento em que um vértice termina de ser processado (todos seus vizinhos já foram visitados), insira esse vértice no final de uma lista.
3. Retorne a lista em ordem reversa.

```
def ordenacao_topologica(grafo):
    """
    DFS modificada para retornar a ordenação topológica do grafo.
    Isso é feito por meio da lista com os tempos de término do processamento
    de cada vértice, que será a ordenação topológica reversa do grafo.
    """
    def dfs_recursiva(grafo, vertice):
        visitados.add(vertice)
        for vizinho in grafo[vertice]:
            if vizinho not in visitados:
                dfs_recursiva(grafo, vizinho)
        tempo += 1
        ordem_topologica[vertice] = tempo

    visitados = set()
    ordem_topologica = [0] * len(grafo)
    tempo = 0
    for vertice in grafo:
        if not vertice in visitados:
            dfs_recursiva(grafo, vertice)
    ordem_topologica.reverse()
    return ordem_topologica

# Ordenação topológica.
ordem = ordenacao_topologica(grafo)
```

Note que na definição do problema dissemos que cada vértice recebe um número de 00 a n−1n−1, mas em nossa

implementação retornamos uma lista com os identificadores dos vértices. Isso é mais conveniente em termos de implementação e uso da lista retornada. Entretanto, podemos pensar nos números de 0 a $n-1$ mencionados na definição como sendo os índices da lista retornada.

Implementação Baseada no Grau dos Vértices

Existe ainda um outro algoritmo de ordenação topológica cuja ideia central é remover do grafo vértices com grau de entrada igual a zero. Esse algoritmo é bastante elegante e nos ensina a raciocinar de forma indutiva a respeito do grafo. Vejamos como ele funciona.

A ideia deste algoritmo é a seguinte:

1. Encontre um vértice com grau de entrada zero.
2. Atribua a esse vértice o menor número disponível (ou coloque-o ao final de uma lista, como iremos fazer) ainda não utilizado.
3. Remova o vértice do grafo e atualize o grau de entrada de todos os vizinhos do vértice removido.
4. Repita os passos acima até que todos os vértices do grafo tenham sido numerados (colocados na lista).

Os maiores desafios do algoritmo acima são os passos 1 e 3. Antes de partirmos para a implementação do algoritmo, é importante discutir esses desafios.

A primeira pergunta interessante com relação ao passo 1 é a seguinte: É sempre possível encontrar um vértice com grau de entrada zero?

Um outro jeito de responder a pergunta anterior é dizer que *todo grafo acíclico direcionado (DAG) tem um vértice de origem (um vértice com grau de entrada zero)*. O raciocínio por trás dessa afirmação é que, dado um vértice qualquer, se você seguir cada aresta de entrada dele em ordem reversa, você eventualmente

alcançará o vértice de origem. Caso contrário, você voltaria ao vértice do qual você partiu, ou seja, você encontraria um ciclo no grafo, o que é impossível, dado que o grafo é um DAG.

A segunda pergunta interessante é: Como calcular o grau de entrada dos vértices? Para fazer isso, podemos percorrer cada vértice do grafo (de forma parecida com uma busca em profundidade ou com uma busca em largura) e atualizar o grau de entrada dos vizinhos do vértice em questão. Por exemplo, podemos fazer isso como mostrado abaixo:

```
graus_entrada = [0 for _ in range(len(grafo))]
for vertice in grafo:
    for vizinho in grafo[vertice]:
        graus_entrada[vizinho] += 1
```

Existem também alguns aspectos interessantes com relação ao passo 3. Esse passo nos pede para remover o vértice do grafo e atualizar o grau de entrada dos vizinhos desse vértice. Como acontece com frequência ao implementarmos algoritmos, por questões de eficiência, não nos preocuparemos em reproduzir *exatamente* o que foi dito nesse passo, mas encontraremos uma forma alternativa de implementar a ideia que ele transmite.

Essa forma alternativa é a seguinte. Manteremos uma fila de vértices com grau de entrada igual a zero. Com isso, em vez de removermos o vértice do grafo, simplesmente o removeremos dessa fila e, ao fazer isso, atualizaremos o grau de entrada de seus vizinhos (basta decrementar de uma unidade o grau de entrada de todos os vizinhos do vértice). O algoritmo executará enquanto houver vértices nessa fila (o passo 4 muda um pouco também).

Com essas discussões, podemos refinar um pouco a ideia do algoritmo:

1. Calcule o grau de entrada de todos os vértices do grafo e armazene esses graus de entrada em uma lista.

2. Percorra a lista e insira em uma fila todos os vértices com grau de entrada igual a zero (conforme vimos, haverá pelos menos um vértice com essa propriedade).
3. Remova o primeiro vértice da fila, insira-o em uma lista que armazena a ordem topológica do grafo e atualize o grau de entrada de seus vizinhos. Se nesse momento o grau de entrada de algum dos vizinhos do vértice em questão se tornar zero, insira esse vizinho na fila.
4. Repita os passos acima enquanto houver vértices na fila.

Agora estamos prontos para implementar o algoritmo.

```
def ordenacao_topologica(grafo):
    """
    Ordenação topológica baseada no grau de entrada dos vértices.
    """
    ordem_topologica = []
    # Calcula graus de entrada.
    graus_entrada = [0 for _ in range(len(grafo))]
    for vertice in grafo:
        for vizinho in grafo[vertice]:
            graus_entrada[vizinho] += 1
    # Cria uma fila de vértices com grau de entrada zero.
    fila = [v for v in range(len(grafo)) if graus_entrada[v] == 0]
    while fila:
        vertice = fila.pop()
        ordem_topologica.append(vertice)
        # Atualiza o grau de entrada dos vizinhos.
        for vizinho in grafo[vertice]:
            graus_entrada[vizinho] -= 1
            # Algum dos vizinhos passou a ter grau de entrada zero.
            if graus_entrada[vizinho] == 0:
                fila.append(vizinho)
    return ordem_topologica
```