
FIAP

PYTHON – AULA 7
LÓGICA DE PROGRAMAÇÃO APLICADA

AGENDA

Aula 1	Instalando e conhecendo o Python e as ferramentas do curso. Revisitando lógica de programação em Python.
Aula 2	Revisitando lógica de programação em Python – continuação.
Aula 3	Trabalhando com listas, tuplas e dicionários. Leitura e escrita de JSON.
Aula 4	Leitura e escrita de arquivos. Introdução a Numpy e Pandas.

AGENDA

Aula 5	Orientação a objeto no Python.
Aula 6	Trabalhando com os algoritmos. Ordenação e Recursão. – Pesquisa em largura.
Aula 7	Algoritmo de Dijkstra.
Aula 8	Algoritmos Genéticos.

PROGRAMANDO EM PYTHON

GRAFOS

ALGORITMO DE DIJKSTRA

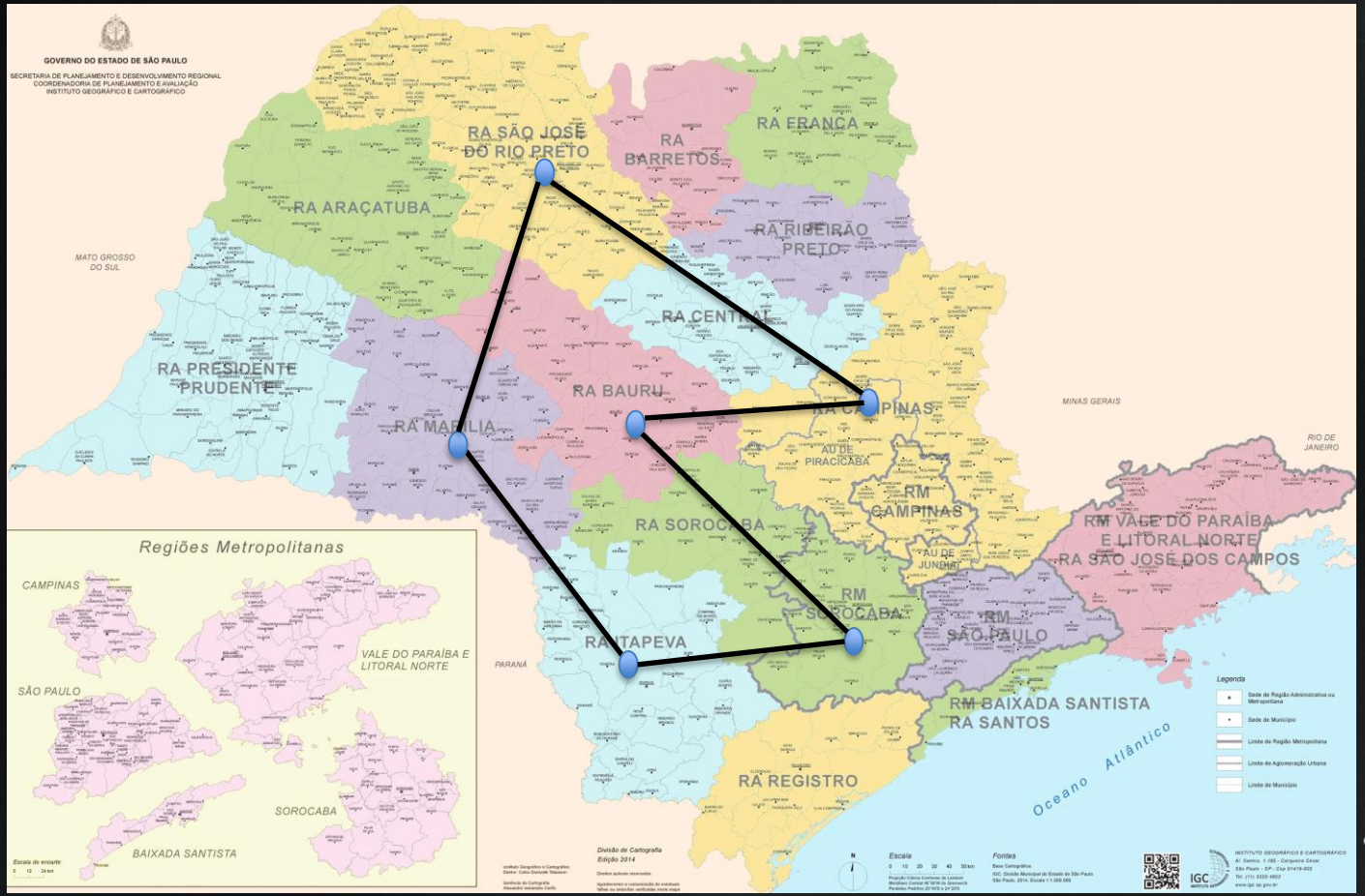
Introdução

- Existe um problema famoso na ciência da computação denominado caixeiro-viajante. A ideia é que um caixeiro precisa visitar um número *n* de cidades percorrendo a distância mínima para visitar toda elas.

- Se considerarmos um total de 6 cidades, por exemplo, devemos analisar 720 permutações (6!), ou seja, 720 operações.

Extraído de : https://br.pinterest.com/pin/37084396909133260

Introdução



Possível rota para o caixeiro-viajante entre cidades do estado de São Paulo.

Introdução

- O algoritmo do caixeiro-viajante tem uma performance muito ruim, crescendo de forma drástica e tornando inviável sua computação (problema intratável) acima de um número muito grande de cidades.

- Por exemplo, 100 cidades equivalem a 10^{158} opções; considerando 1 bilhão de soluções por segundo, levaria 10^{140} anos para encontrar a melhor (mais tempo do que a idade do universo).

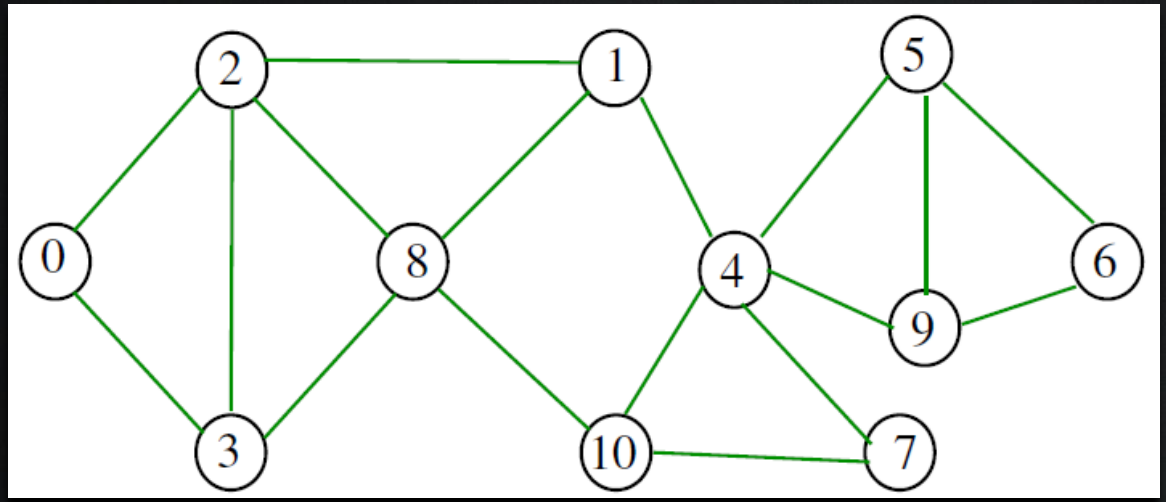
Introdução

- Apesar do algoritmo do caixeiro-viajante ter uma performance ruim e ser um problema intratável para um grande número de cidades, ele nos fornece uma abordagem interessante ao mostrar conexões.

- Um conjunto de conexões são denominadas de grafos e a partir desta abordagem iremos conhecer algoritmos que irão nos oferecer uma solução aproximada em tempo hábil para o problema do caixeiro-viajante e muitos outros de aplicações reais

Grafos

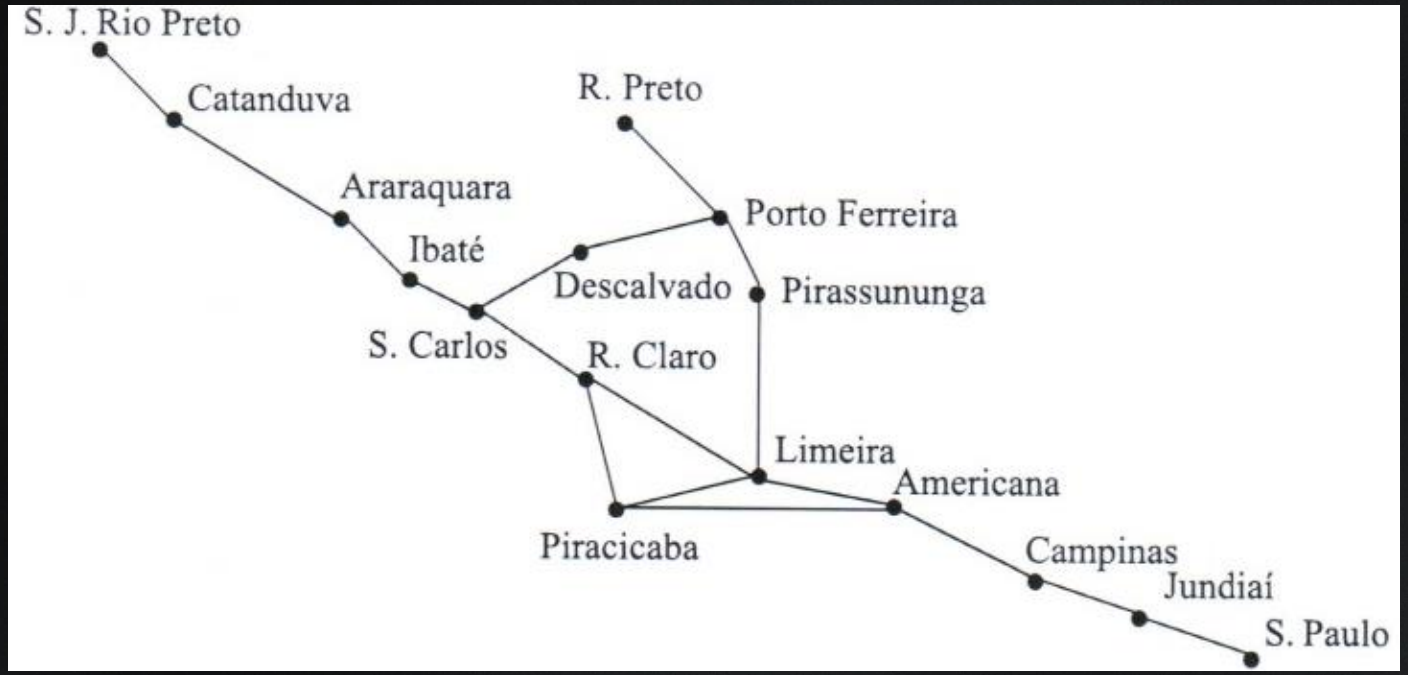
Um grafo é um conjunto de conexões, onde temos os nós ou vértices (N) e as arestas ou arcos (A). Trata-se de um modelo ou representação da realidade e nos ajuda com simulações.



Exemplo de grafo

Aplicações

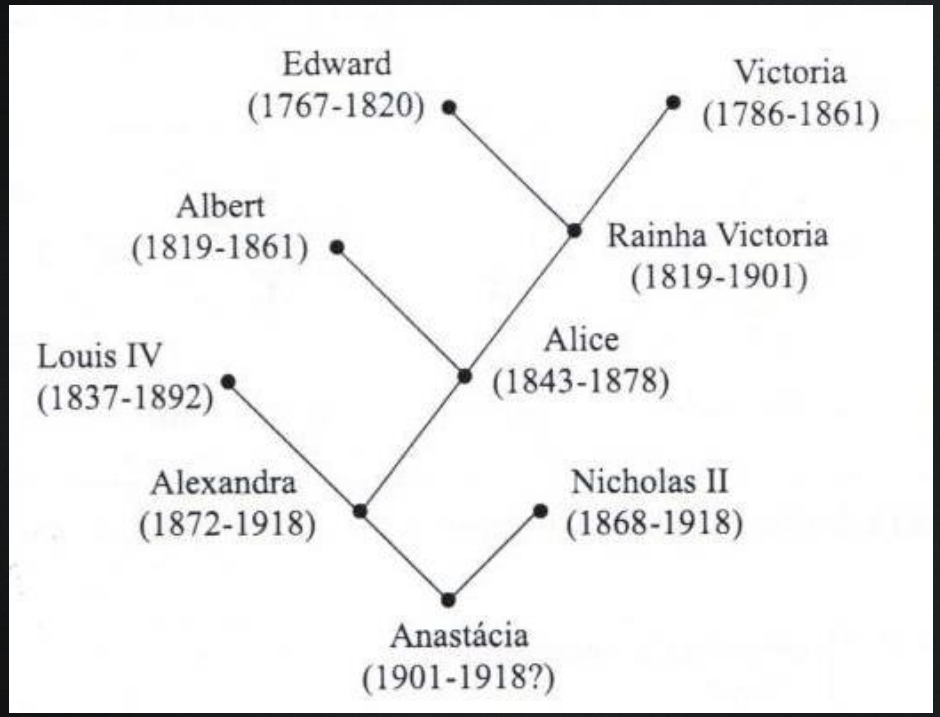
- Podem ser usados para representar mapas.



Grafo que representa o mapa parcial de uma região do Estado de São Paulo. [Extraído de NICOLETTI].

Aplicações

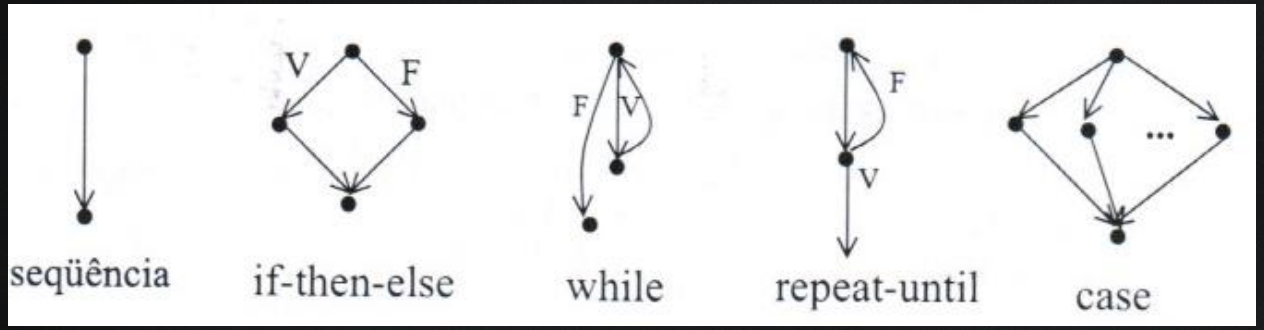
- Podem ser usados para representar Árvore Genealógica



Grafo que representação de relações familiares. [Extraído de NICOLETTI].

Aplicações

Pode ser aplicado na computação, por exemplo na geração de casos de teste.

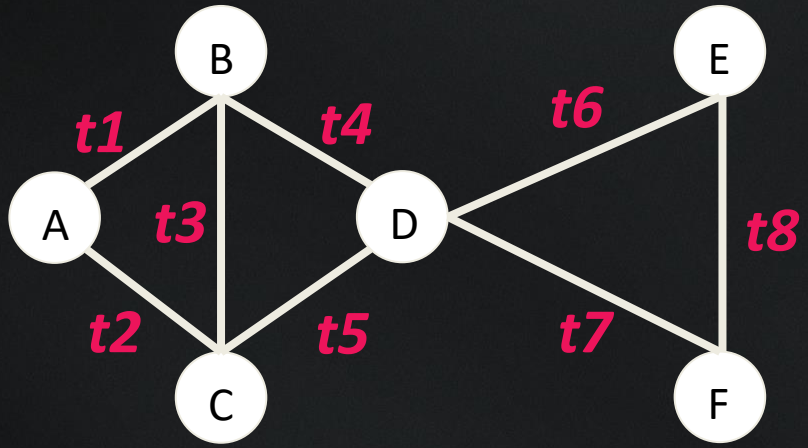


Notação de vários comandos usando grafos de fluxo. [Extraído de NICOLETTI].

Exemplo de teste de controle: grafo de fluxo de um módulo que gere conjunto de caminhos independentes que devem ser percorridos para garantir que todos os comandos (e ramificações) sejam executados pelo menos uma vez.

Aplicações

Podem ser usados para modelar uma rede telefônica.



Modelando uma rede telefônica. [Extraído e adaptado de NICOLETTI].

Exercício

1. Onde mais podemos encontrar aplicações de grafos no mundo real ?
2. Quais empresas utilizam grandes grafos otimizados?

1. Fornecimento de:
    - Eletricidade;
    - Gás;
    - Água.
    - Rádio;
    - Tarifas áreas;
2. Vendas
3. Logística
4. Supply Chain
5. GPS

FedEx®

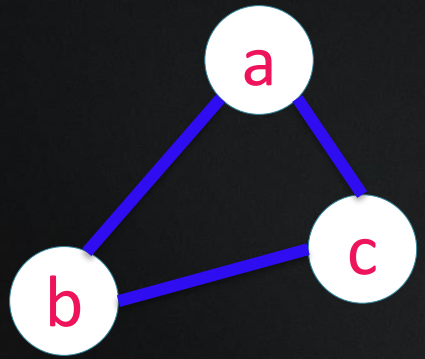
Walmart
Save money. Live better.

Grafo Dirigido

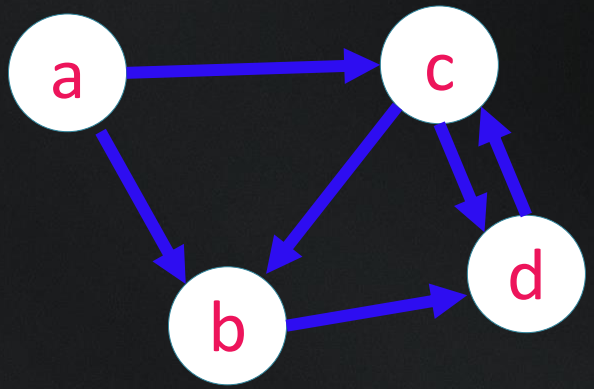
- É uma dupla (N, A), que se pode definir a ordem nos pares de nós

Grafo Não Dirigido

- Não pode definir a ordem nos pares de nós.

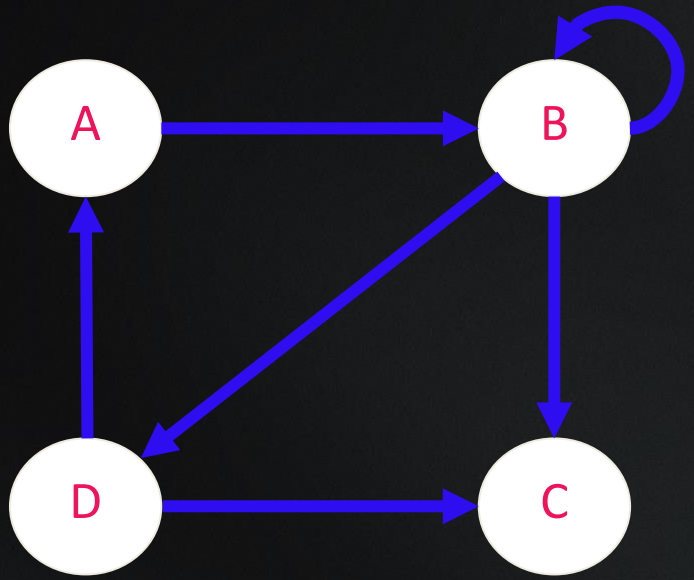


Grafo não dirigido



Grafo dirigido

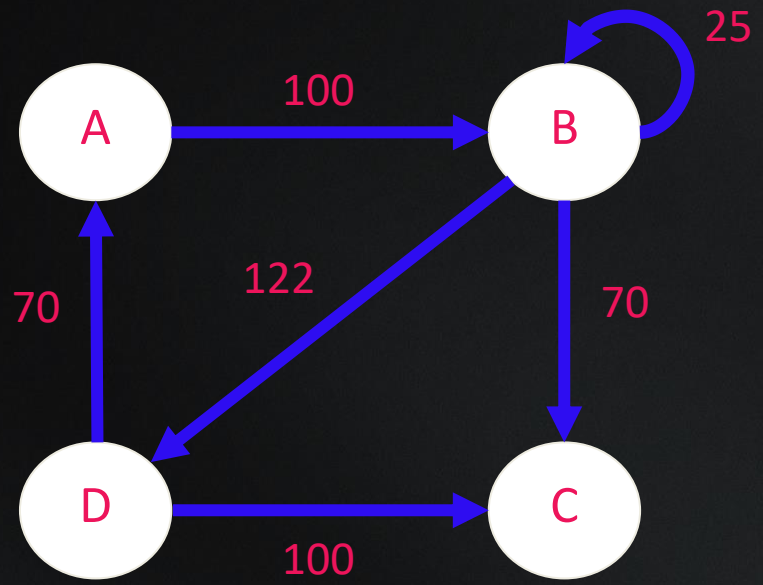
Grau de um Nó



- O nó A tem grau 2 = uma saída mais uma entrada;
- O nó B tem grau 5 (3 + 2) = três saídas mais duas entradas.

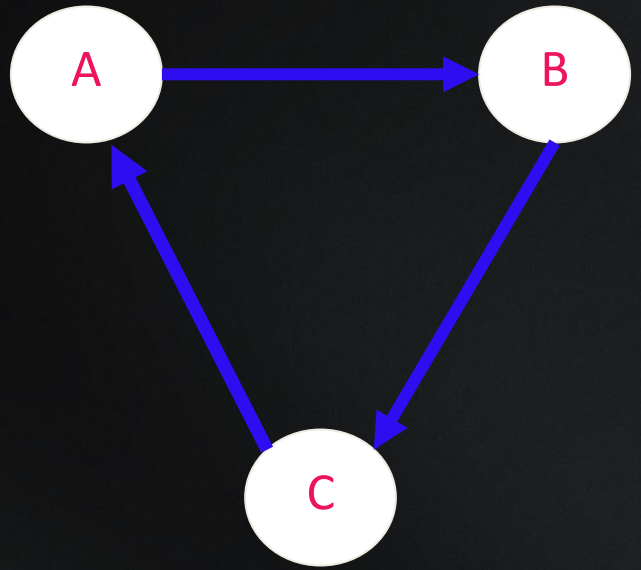
Pesos das arestas

O peso dos arcos representa o grau de importância na conexão entre dois nós.



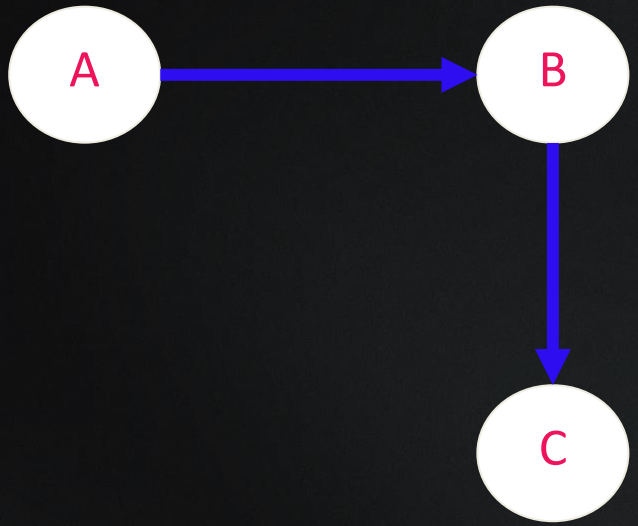
No caso de mapas, pode-se considerar a distância em quilômetros ou milhas.

Grafo Cíclico



Existe um caminho de no mínimo três “nós” que começam e terminam no mesmo “Nó”.

Grafo Acíclico



Não ocorre nenhum ciclo em circuito nos “Nós”.

Representação (construção) do grafo

- Os conjuntos G=(N,A) podem ser representados pelas estruturas de dados:

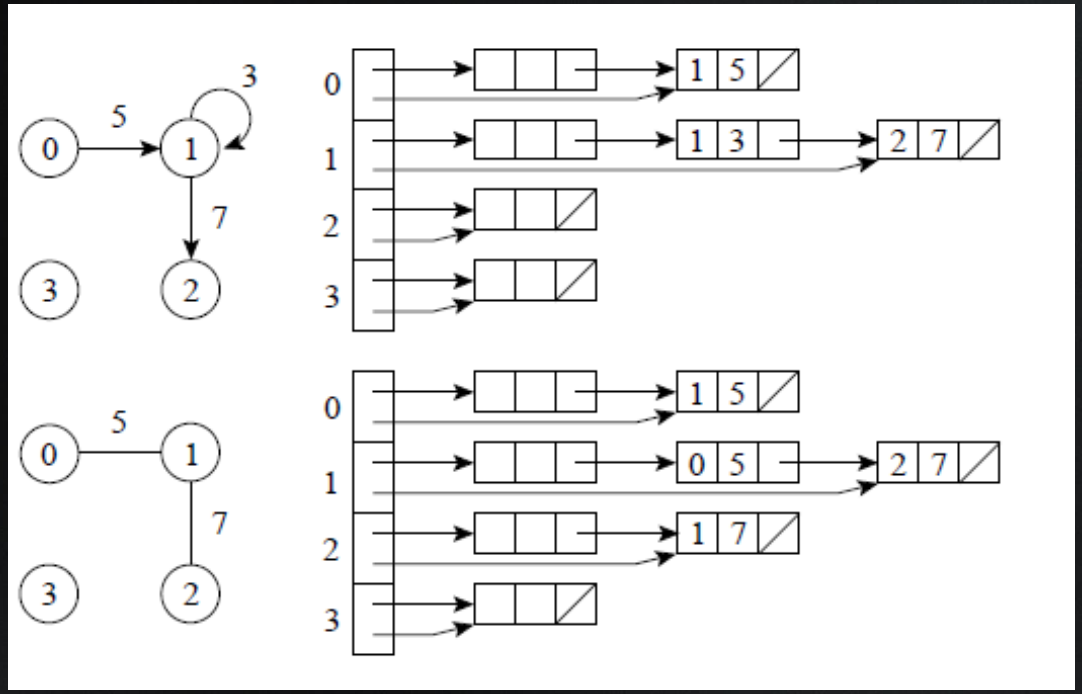
a) Lista de adjacência;
b) Matriz de adjacência.

Algoritmo

- O processo para percorrer o grafo de uma forma simples denomina-se **busca em largura.**

Lista de Adjacência

- Usa um vetor com *n* **listas ligadas (filas)**. Cada posição do vetor corresponde a um Nó de G(N,A), e os arcos de um certo Nó para outros Nós são representados por listas ligadas.



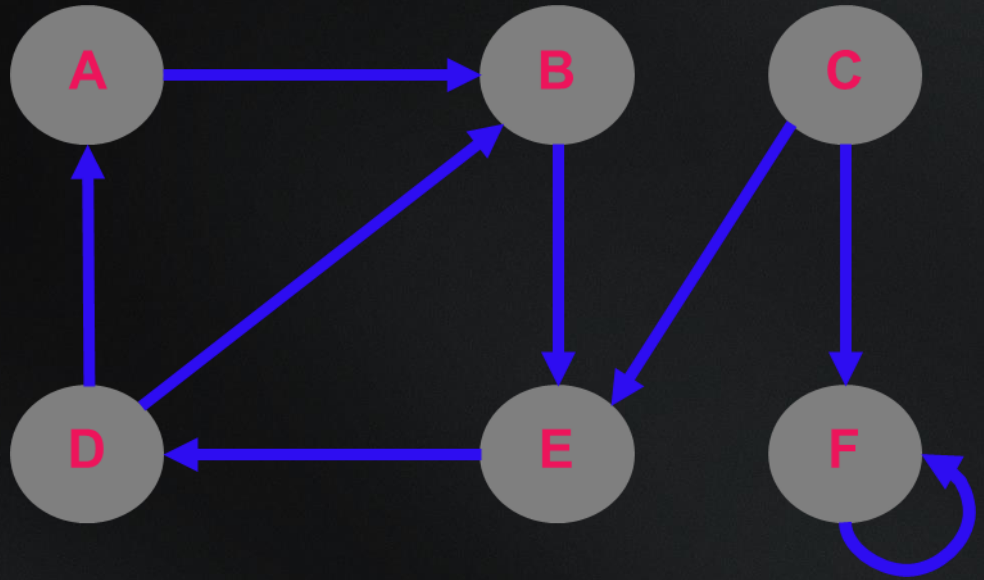
Lista de adjacência [Extraído de ALMEIDA]

Filas são estruturas de dados (semelhante a filas na vida real), onde o primeiro elemento a ser inserido, será também o primeiro a ser retirado (FIFO).

Remoção - dequeue.
Inserção - enqueue.

Matriz de Adjacência

- Assume que os Nós são numerados de 1 até N. A matriz é construída com dimensão NxN de elementos e_{ij}.



	A	B	C	D	E	F
A	0	1	0	0	0	0
B	0	0	0	0	1	0
C	0	0	0	0	1	1
D	1	1	0	0	0	0
E	0	0	0	1	0	0
F	0	0	0	0	0	1

Grafo e Matriz [Extraído e Adaptado de Silva]

Busca em Largura

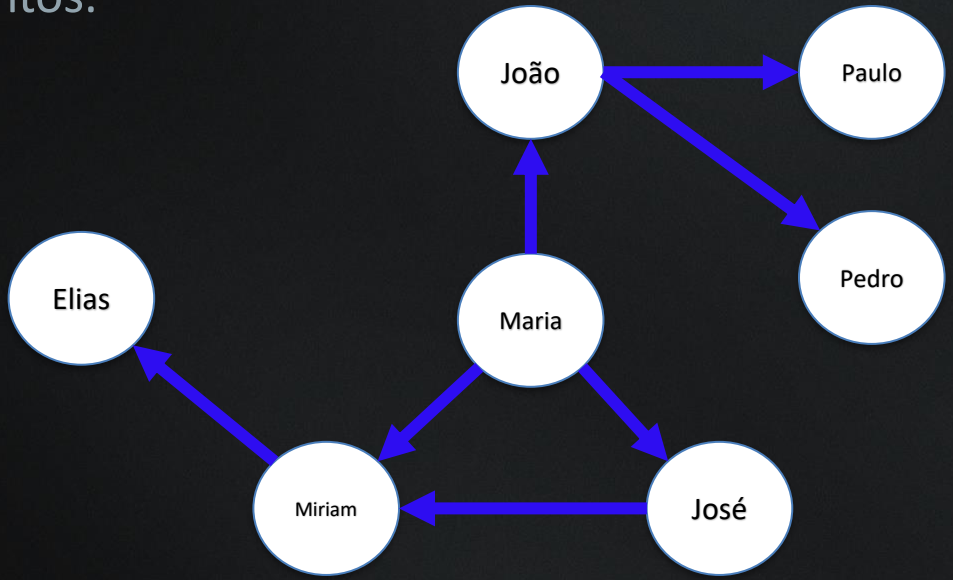
- A pesquisa em largura busca pelo caminho mínimo, ou seja, o mínimo de arestas a serem percorridas para chegar ao objetivo final.

- A busca começa por um vértice especificado pelo usuário, depois consulta os vértices vizinhos e todos os vizinhos dos vizinhos.

- O algoritmo numera os vértices na ordem em que eles são descobertos; para fazer isso, o algoritmo usa uma **fila** de vértices.

Implementação do grafo

- Para implementar nosso primeiro código, iremos utilizar estruturas de dados, tais como listas e dicionários. Considere o grafo abaixo e seus relacionamentos.



- Queremos saber se nessa rede de relacionamentos existe algum pescador

Implementação do grafo

Para o grafo vamos criar um dicionário: grafo = {}

E popular por meio da chave – valor:

grafo[“maria”] = [“joão”, “miriam”, “jose”]
grafo[“joão”] = [“paulo”, “pedro”]
grafo[“jose”] = [“miriam”]
grafo[miriam] = [“elias”]

Implementação do algoritmo

- Para implementar o algoritmo, devemos:

1. criar uma fila de todas as pessoas que serão verificadas (comece pelo centro);
2. retirar uma pessoa da fila e verificar se é um pescador;
3. caso seja, encontramos o que precisamos, se não, devemos adicionar todos os vizinhos dessa pessoa na fila; e
4. repetir o passo 2.

Implementação em Python

```python
from collections import deque

grafo = {}
grafo["maria"] = ["joão", "miriam", "jose"]
grafo["joão"] = ["paulo", "pedro"]
grafo["jose"] = ["miriam"]
grafo["miriam"] = ["elias"]
grafo["paulo"] = []
grafo["pedro"] = []
grafo["elias"] = []
```

Utilizamos a biblioteca *deque* de *collections* que cria a fila.

Implementação em Python

```python
def pesquisa(nome):
    fila = deque()
    fila += grafo[nome]
    verificadas = []
    while fila:
        pessoa = fila.popleft()
        if not pessoa in verificadas:
            if pescador(pessoa):
                print(pessoa + " é pescador")
                return True
            else:
                fila += grafo[pessoa]
                verificadas.append(pessoa)
    print("Não existem pescadores")
    return False


def pescador(nome):
    return nome[-1] == 's' #verifica se o nome da pessoa termina com a letra especificada

pesquisa("jose")
```

A função popleft() remove a esquerda da lista

Exercícios

1. Modifique a função pescador para algo mais interessante e veja se representa o que você espera.

Algoritmo de Dijkstra

- O algoritmo de Dijkstra determina o caminho mínimo até um objetivo em um grafo ponderado.

- Vimos que para percorrer um grafo de um ponto a outro utilizamos busca em largura por meio de listas ligadas.

Algoritmo de Dijkstra

- A busca em largura não é necessariamente o caminho mais rápido, mas é o mais curto. Entretanto, suponha que seja adicionado tempo de deslocamento aos segmentos (arestas).

- Nesse caso, com o algoritmo de Dikstra podemos calcular o caminho mais rápido.

- Este algoritmo resolve o problema do caminho mínimo em uma rede com múltiplas possibilidades, dado um ponto de partida.

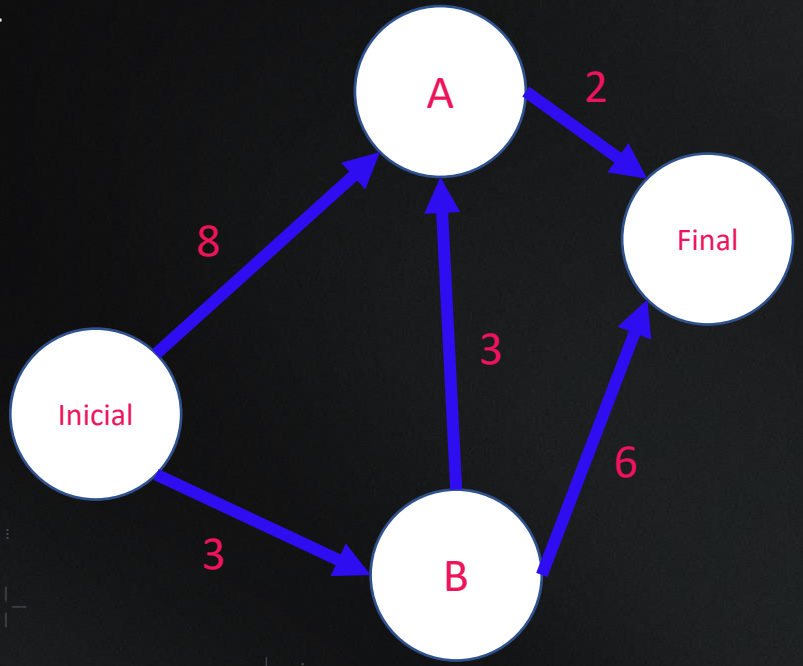
Algoritmo de Dijkstra

- O algoritmo de Dijkstra explora continuamente os vértices mais próximos ao inicial; quando encontra um novo, ele armazena a distancia do inicial até esse último e atualiza caso encontre um caminho mais curto.

- Além disso, o algoritmo também armazena qual aresta levou a cada vértice.

Implementação do algoritmo Dijkstra

- A partir do ponto inicial, você está pensando se deve ir ao ponto A ou B, considerando o tempo para alcançar cada um, e assim atingir o objetivo (ponto final).



1. O caminho para a aresta B leva menos tempo e portanto, você escolhe ir por ele.
2. Ao chegar em B, você percebe que o caminho para A só leva 3 min, logo, você encontrou um caminho mais curto para o ponto A.
3. Por fim o caminho até o final leva 8 minutos.

Obs: a busca em largura teria optado pelo caminho mais curto, ou seja, de inicial para A e deste para o final (duas arestas), consumindo 10 minutos.

Implementação do algoritmo Dijkstra

1. Coloque o vértice inicial em uma fila de prioridades
2. Remova o vértice mais próximo da fila.
3. Verifique todos os vizinhos conectados ao nó atual; se não foram registrados ou se a aresta propor um novo caminho mínimo, para cada um, registre sua distância a partir do início, armazene a aresta que proporcionou essa distância e acrescente o novo vértice a fila de prioridades.
4. Repita os passos 2 e 3 até esvaziar a fila
5. Retorne a distância mínima para todos os vértices a partir do inicial, e o caminho para cada um.

Implementação do algoritmo Dijkstra em Python

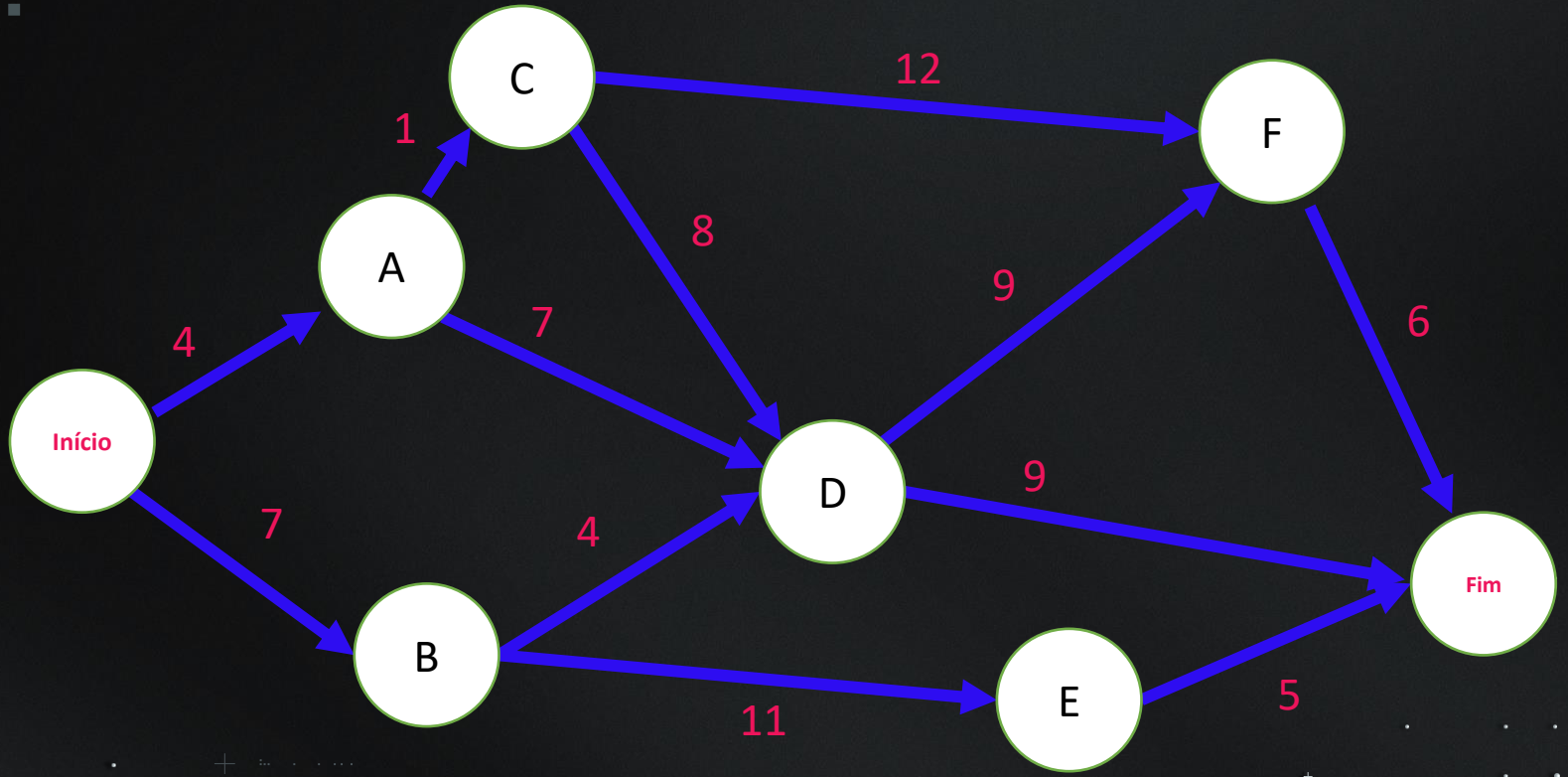
```python
#construção do grafo
grafo = {}

grafo["inicial"] = {}
grafo["inicial"]["a"] = 8
grafo["inicial"]["b"] = 3
grafo["a"] = {}
grafo["a"]["final"] = 2
grafo["b"] = {}
grafo["b"]["a"] = 3
grafo["b"]["final"] = 6
grafo["final"] = {}

infinito = float("inf") #representação de infinito em python
custos = {}
custos["a"] = 8
custos["b"] = 3
custos["final"] = infinito

pais = {}
pais["a"] = "inicial"
pais["b"] = "inicial"
pais["final"] = None

processados = []
```

Implementação do algoritmo Dijkstra em python

```python
#algoritmo
def achar_melhor_custo(custos):
    custo_baixo = float("inf")
    vertice_melhor_custo = None
    for vertice in custos:
        custo = custos[vertice]
        if custo < custo_baixo and vertice not in processados:
            custo_baixo = custo
            vertice_melhor_custo = vertice
    return vertice_melhor_custo

vertice = achar_melhor_custo(custos)

while vertice is not None:
    custo = custos[vertice]
    vizinhos = grafo[vertice]
    for n in vizinhos.keys():
        novo_custo = custo + vizinhos[n]
        if custos[n] > novo_custo:
            custos[n] = novo_custo
            pais[n] = vertice
    processados.append(vertice)
    vertice = achar_melhor_custo(custos)

print(custo)
```

Exercícios

Encontre o melhor custo (caminho mínimo) do início ao fim



Revisando

- Pesquisa em largura é usada para calcular o caminho mínimo em um grafo não ponderado.

- Algoritmo de Dijkstra é usado para calcular caminho mínimo em um grafo ponderado, porém com os pesos positivos.

- Para problemas que requerem muitas variáveis (vértices), usaremos algoritmos de aproximação (genéticos).

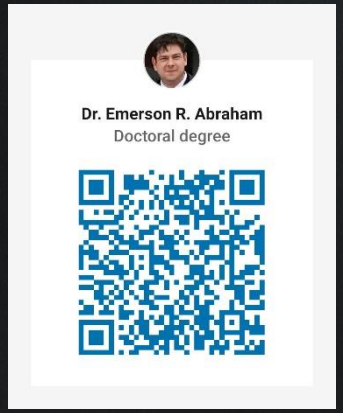
BIBLIOGRAFIA BÁSICA

- BEAZLEY, David. Python Essential Reference,. 2009.
- BHARGAVA, ADITYA Y. Entendendo Algoritmos. Um guia ilustrado para programadores e outros curiosos. São Paulo: Ed. Novatec, 2017
- CORMEN, THOMAS H. et al. Algoritmos: teoria e prática. Rio de Janeiro: Elsevier, 2002
- COSTA, Sérgio Souza. Recursividade. Professor Adjunto da Universidade Federal do Maranhão.
- DOWNEY, ALLEN B. Pense em Python. Pense como um cientista da computação. São Paulo: Ed. Novatec, 2016
- GRANATYR, Jones; PACHOLOK, Edson. IA Expert Academy. Disponível em: https://iaexpert.academy/
- KOPEC, DAVID. Problemas clássicos de ciência da computação com Python. São Paulo: Ed. Novatec, 2019
- LINDEN, Ricardo. Algoritmos Genéticos. 3 edição. Rio de Janeiro: Editora Moderna, 2012
- MCKINNEY, WILLIAM WESLEY. Python para análise de dados. Tratamento de dados com Pandas, Numpy e Ipython. São Paulo: Ed. Novatec, 2018

BIBLIOGRAFIA BÁSICA

- TENEMBAUM, Aaron M. Estrutura de Dados Usando C. Sao Paulo: Makron Books do Brasil, 1995.
- VELLOSO, Paulo. Estruturas de Dados. Rio de Janeiro: Ed. Campus, 1991.
- VILLAS, Marcos V & Outros. Estruturas de Dados: Conceitos e Tecnicas de implementacao. RJ: Ed. Campus, 1993.
- PREISS, Bruno P. Estrutura de dados e algoritmos: Padrões de projetos orientados a objetos com Java. Rio de Janeiro: Editora Campus, 2001.
- PUGA, Sandra; RISSETTI, Gerson. Lógica de programação e estruturas de dados. 2016.
- SILVA, Osmar Quirino. Estrutura de Dados e Algoritmos Usando C. Fundamentos e Aplicações. Rio de Janeiro: Editora Ciência Moderna, 2007.
- ZIVIANI, N. Projeto de algoritmos com implementações em pascal e C. São Paulo: Editora Thomsom, 2002.

OBRIGADO



FIAP


FIAP