

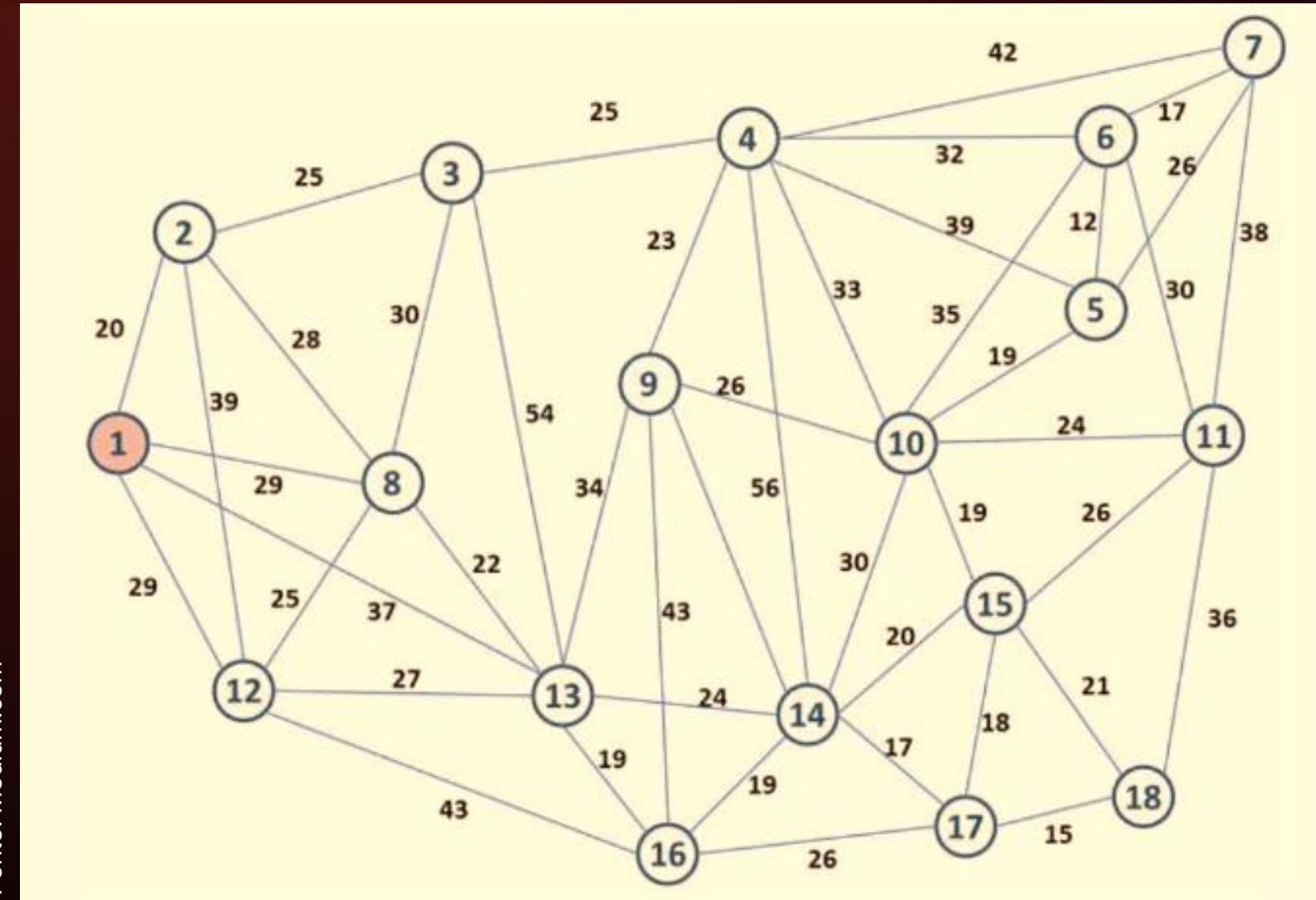
APRENDIZADO DE MÁQUINAS

Algoritmos Genéticos
Implementação do problema
do caixeiro viajante

TÓPICOS

1. Introdução
2. O problema do Caixeiro Viajante
3. Métodos de solução
4. Algoritmo em Python

PROBLEMA DO CAIXEIRO VIAJANTE



PROBLEMA DO CAIXEIRO VIAJANTE

O Desafio

Em 1962, a Procter & Gamble lançou um desafio com um prêmio de US\$10.000,00 (o suficiente para comprar uma casa na época) e 54 prêmios de US\$ 1.000,00 para quem resolvesse um PCV com 33 cidades espalhadas pelos EUA.

A solução ótima é uma rota de ≈ 17480 km de comprimento, e vários participantes a alcançaram, mesmo sem saber que esta era a solução ótima. O critério de desempate foi a melhor redação sobre produtos da Procter & Gamble.



PROBLEMA DO CAIXEIRO VIAJANTE

Soluções:

- Métodos exatos
- Métodos heurísticos
 - Exemplos:
 - Vizinho mais próximo
 - Cobertura mínima (Dijkstra)
 - Algoritmos Genéticos

PROBLEMA DO CAIXEIRO VIAJANTE

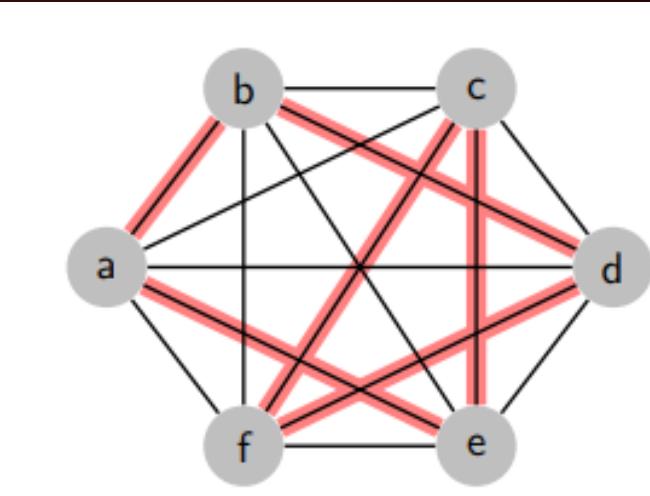
Indivíduos:

O cromossomo será constituído de um vetor de valores (de 1 a n), que corresponde ao número de cada cidade.

Função de avaliação: minimizar o valor

$f(x) = \text{soma das distâncias entre as cidades}$

A seleção da população será realizada por meio do fitness.



	a	b	c	d	e	f
a	-	1	2	3	1	2
b	1	-	7	1	4	3
c	2	7	-	3	1	1
d	3	1	3	-	8	1
e	1	4	1	8	-	2
f	2	3	1	1	2	-

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [12]: # Imports
import numpy as np
import random

from datetime import datetime

In [13]: #quantidade = int(input("Digite a quantidade de cidades (+ que duas): "))
#while quantidade < 2:
#    print("Quantidade invalida!!!")
#    quantidade = int(input("Digite a quantidade de cidades (+ que duas): "))
#tamanhoPopulacao = int(input("Digite o tamanho da população (+ que um):"))
#while tamanhoPopulacao < 1:
#    print("Quantidade invalida!!!")
#    tamanhoPopulacao = int(input("Digite o tamanho da população (+ que um):"))
#geracoes = int(input("Digite a quantidade de gerações (+ que uma): "))
#while geracoes < 1:
#    print("Quantidade invalida!!!")
#    geracoes = int(input("Digite a quantidade de gerações (+ que uma): "))

# Parameters
n_cities = 6

n_population = 20

mutation_rate = 0.3
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [18]: # Generating a list of coordenades representing each city
coordinates_list = [[x,y] for x,y in zip(np.random.randint(0,100,n_cities),np.random.randint(0,100,n_cities))]
names_list = np.array(['CidadeA', 'CidadeB', 'CidadeC', 'CidadeD', 'CidadeE', 'CidadeF'])
cities_dict = { x:y for x,y in zip(names_list,coordinates_list)}

# Function to compute the distance between two points
def compute_city_distance_coordinates(a,b):
    return ((a[0]-b[0])**2+(a[1]-b[1])**2)**0.5

def compute_city_distance_names(city_a, city_b, cities_dict):
    return compute_city_distance_coordinates_fixo(cities_dict[city_a], cities_dict[city_b])

def compute_city_distance_coordinates_fixo(a,b):
    if a == 'CidadeA' and b == 'CidadeB':
        return 1
    if a == 'CidadeA' and b == 'CidadeC':
        return 2
    if a == 'CidadeA' and b == 'CidadeD':
        return 3
    if a == 'CidadeA' and b == 'CidadeE':
        return 1
    if a == 'CidadeA' and b == 'CidadeF':
        return 2
    if a == 'CidadeB' and b == 'CidadeA':
        return 1

def compute_city_distance_names_fixo(city_a, city_b, cities_dict):
    return compute_city_distance_coordinates_fixo(city_a, city_b)

cities_dict

Out[18]: {'CidadeA': [83, 66],
'CidadeB': [55, 21],
'CidadeC': [74, 2],
'CidadeD': [95, 58],
'CidadeE': [33, 56],
'CidadeF': [9, 62]}
```

Entrada da Dados:
distância entre as
cidades (matriz
adjacências)



PROBLEMA DO CAIXEIRO VIAJANTE

```
In [30]: # First step: Create the first population set
def genesis(city_list, n_population):

    population_set = []
    for i in range(n_population):
        #Randomly generating a new solution
        sol_i = city_list[np.random.choice(list(range(n_cities)), n_cities, replace=False)]
        population_set.append(sol_i)
    return np.array(population_set)

population_set = genesis(names_list, n_population)
population_set
```

```
Out[30]: array([['CidadeC', 'CidadeD', 'CidadeE', 'CidadeB', 'CidadeA', 'CidadeF'],
   ['CidadeE', 'CidadeC', 'CidadeA', 'CidadeF', 'CidadeB', 'CidadeD'],
   ['CidadeE', 'CidadeB', 'CidadeD', 'CidadeA', 'CidadeC', 'CidadeF'],
   ['CidadeE', 'CidadeB', 'CidadeC', 'CidadeF', 'CidadeD', 'CidadeA'],
   ['CidadeB', 'CidadeA', 'CidadeF', 'CidadeD', 'CidadeC', 'CidadeE'],
   ['CidadeB', 'CidadeC', 'CidadeA', 'CidadeE', 'CidadeD', 'CidadeF'],
   ['CidadeF', 'CidadeC', 'CidadeD', 'CidadeB', 'CidadeE', 'CidadeA'],
   ['CidadeA', 'CidadeB', 'CidadeD', 'CidadeE', 'CidadeF', 'CidadeC'],
   ['CidadeF', 'CidadeC', 'CidadeB', 'CidadeA', 'CidadeE', 'CidadeD'],
   ['CidadeB', 'CidadeC', 'CidadeF', 'CidadeE', 'CidadeD', 'CidadeA'],
   ['CidadeE', 'CidadeB', 'CidadeA', 'CidadeC', 'CidadeF', 'CidadeD'],
   ['CidadeC', 'CidadeB', 'CidadeA', 'CidadeF', 'CidadeD', 'CidadeE'],
   ['CidadeB', 'CidadeA', 'CidadeF', 'CidadeD', 'CidadeC', 'CidadeE'],
   ['CidadeC', 'CidadeF', 'CidadeD', 'CidadeB', 'CidadeA', 'CidadeE'],
   ['CidadeA', 'CidadeE', 'CidadeC', 'CidadeF', 'CidadeB', 'CidadeD'],
   ['CidadeC', 'CidadeF', 'CidadeE', 'CidadeA', 'CidadeB', 'CidadeD'],
   ['CidadeD', 'CidadeC', 'CidadeF', 'CidadeE', 'CidadeA', 'CidadeB'],
   ['CidadeA', 'CidadeF', 'CidadeD', 'CidadeB', 'CidadeC', 'CidadeE'],
   ['CidadeA', 'CidadeD', 'CidadeB', 'CidadeE', 'CidadeC', 'CidadeF'],
   ['CidadeE', 'CidadeF', 'CidadeB', 'CidadeD', 'CidadeA', 'CidadeC']],
  dtype='|<U7')
```

da
tância
dades
Z
cias)

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [32]: def fitness_eval(city_list, cities_dict):
    total = 0
    for i in range(n_cities-1):
        a = city_list[i]
        b = city_list[i+1]
        #total += compute_city_distance_names(a,b, cities_dict)
        total += compute_city_distance_names_fixo(a,b, cities_dict)
    return total

In [33]: def get_all_fitness(population_set, cities_dict):
    fitness_list = np.zeros(n_population)

    #Looping over all solutions computing the fitness for each solution
    for i in range(n_population):
        fitness_list[i] = fitness_eval(population_set[i], cities_dict)

    return fitness_list

fitness_list = get_all_fitness(population_set,cities_dict)
fitness_list

Out[33]: array([13., 12., 10., 14., 13., 14., 12., 8., 11., 17., 13., 10., 6.,
               16., 6., 7., 14., 12., 19., 21.])
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [34]: def progenitor_selection(population_set,fitnes_list):
    total_fit = fitnes_list.sum()
    prob_list = fitnes_list/total_fit

    #Notice there is the chance that a progenitor. mates with oneself
    progenitor_list_a = np.random.choice(list(range(len(population_set))), len(population_set),p=prob_list, replace=True)
    progenitor_list_b = np.random.choice(list(range(len(population_set))), len(population_set),p=prob_list, replace=True)

    progenitor_list_a = population_set[progenitor_list_a]
    progenitor_list_b = population_set[progenitor_list_b]

    return np.array([progenitor_list_a,progenitor_list_b])

progenitor_list = progenitor_selection(population_set,fitnes_list)
progenitor_list[0][2]
```

```
Out[34]: array(['CidadeF', 'CidadeC', 'CidadeE', 'CidadeA', 'CidadeD', 'CidadeB'],
              dtype='<U7')
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [35]: def mate_progenitors(prog_a, prog_b):
    offspring = prog_a[0:5]

    for city in prog_b:

        if not city in offspring:
            offspring = np.concatenate((offspring,[city]))

    return offspring

def mate_population(progenitor_list):
    new_population_set = []
    for i in range(progenitor_list.shape[1]):
        prog_a, prog_b = progenitor_list[0][i], progenitor_list[1][i]
        offspring = mate_progenitors(prog_a, prog_b)
        new_population_set.append(offspring)

    return new_population_set

new_population_set = mate_population(progenitor_list)
new_population_set[0]

Out[35]: array(['CidadeA', 'CidadeE', 'CidadeF', 'CidadeC', 'CidadeB', 'CidadeD'],
              dtype='|<U7')
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [36]: def mutate_offspring(offspring):
    for q in range(int(n_cities*mutation_rate)):
        a = np.random.randint(0,n_cities)
        b = np.random.randint(0,n_cities)

        offspring[a], offspring[b] = offspring[b], offspring[a]

    return offspring

def mutate_population(new_population_set):
    mutated_pop = []
    for offspring in new_population_set:
        mutated_pop.append(mutate_offspring(offspring))
    return mutated_pop

mutated_pop = mutate_population(new_population_set)
mutated_pop[0]
```

```
Out[36]: array(['CidadeF', 'CidadeE', 'CidadeA', 'CidadeC', 'CidadeB', 'CidadeD'],
              dtype='|<U7')
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [37]: best_solution = [-1,np.inf,np.array([])]
for i in range(10000):
    if i%100==0: print(i, fitnes_list.min(), fitnes_list.mean(), datetime.now().strftime("%d/%m/%y %H:%M"))
    fitnes_list = get_all_fitness(mutated_pop,cities_dict)

    #Saving the best solution
    if fitnes_list.min() < best_solution[1]:
        best_solution[0] = i
        best_solution[1] = fitnes_list.min()
        best_solution[2] = np.array(mutated_pop)[fitnes_list.min() == fitnes_list]

progenitor_list = progenitor_selection(population_set,fitnes_list)
new_population_set = mate_population(progenitor_list)

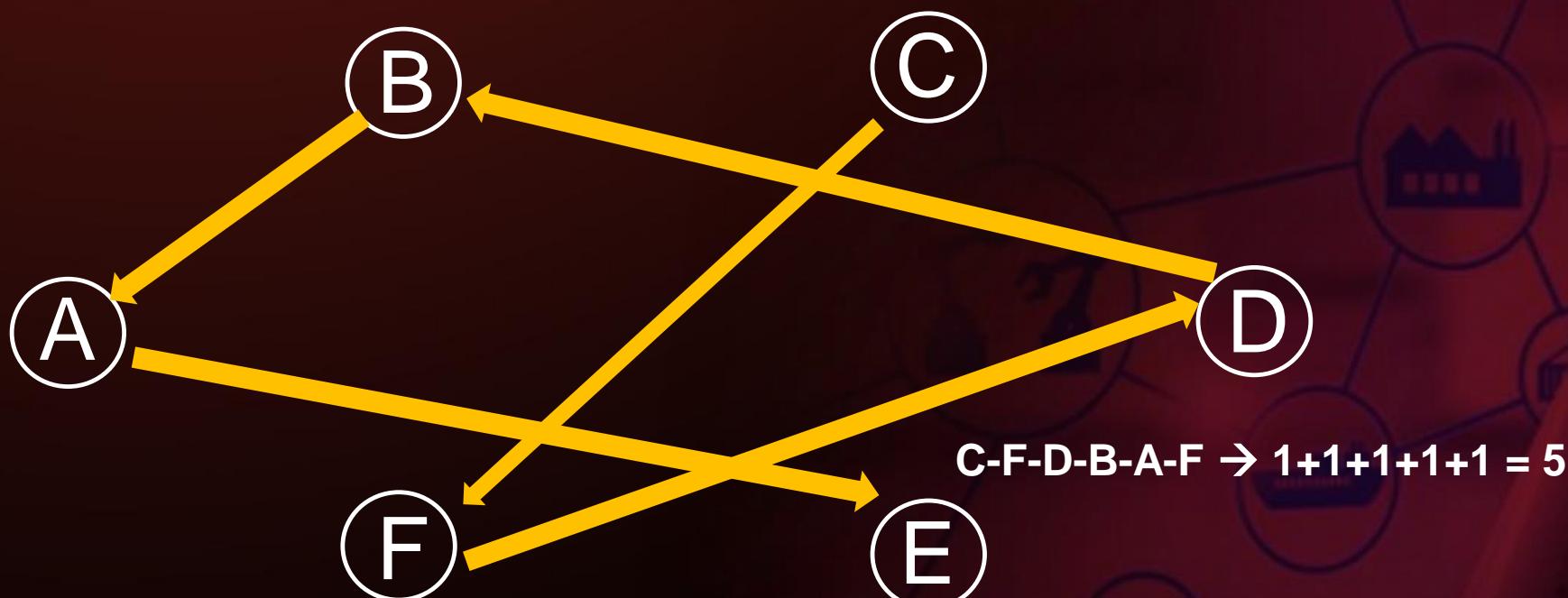
mutated_pop = mutate_population(new_population_set)

0 6.0 12.4 23/10/22 09:43
100 6.0 12.5 23/10/22 09:43
200 7.0 13.05 23/10/22 09:43
300 6.0 12.35 23/10/22 09:43
400 6.0 13.05 23/10/22 09:43
500 7.0 13.35 23/10/22 09:43
600 5.0 12.95 23/10/22 09:43
```

PROBLEMA DO CAIXEIRO VIAJANTE

```
In [38]: best_solution
```

```
Out[38]: [1,
      5.0,
      array([['CidadeC', 'CidadeF', 'CidadeD', 'CidadeB', 'CidadeA', 'CidadeE']],
            dtype='<U7')]
```



APRENDIZADO DE MÁQUINAS

Algoritmos Genéticos
Implementação do problema
do caixeiro viajante