МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ**

**«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСиС»**

Институт ИТКН

Кафедра инженерной кибернетики

Направление подготовки: «01.03.04 Прикладная математика»

Квалификация: бакалавр

Группа: БПМ-17-2

**ОТЧЕТ**

**ПО КУРСОВОЙ РАБОТЕ**

**«ИСКУССТВЕННЫЕ НЕЙРОННЫЕ СЕТИ»**

на тему: «Генетические алгоритмы. Классы задач, решаемые с помощью генетических алгоритмов»

**Студент** \_\_\_\_\_\_\_\_\_\_/ Добрикова Ю.А.

**Руководитель** \_\_\_\_\_\_\_\_\_\_/ старший преподаватель, Кондыбаева А.Б.

**Оценка: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Дата защиты: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Москва 2020**

**Содержание**

[1. Введение 3](#_Toc59327870)

[2. Теоретическая часть 3](#_Toc59327871)

[2.1. Основные понятия 3](#_Toc59327872)

[2.2. Отличия от традиционных алгоритмов 5](#_Toc59327873)

[2.3. Преимущества генетических алгоритмов 6](#_Toc59327874)

[2.4. Базовая структура генетического алгоритма 6](#_Toc59327875)

[2.5. Применение генетических алгоритмов 8](#_Toc59327876)

[3. Реализация модели 9](#_Toc59327877)

[4. Список использованных источников 12](#_Toc59327878)

[5. Приложение 13](#_Toc59327879)

[5.1. Код файла trav.ipynb 13](#_Toc59327880)

[5.2. Код файла tsp.py 14](#_Toc59327881)

# Введение

Г**енетический алгоритм** (genetic algorithm) — это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путём последовательного подбора, комбинирования и вариации искомых параметров с использованием механизмов, напоминающих биологическую эволюцию. Является разновидностью эволюционных вычислений. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

# Теоретическая часть

## 2.1. Основные понятия

**Ген** (также называемый свойством, знаком или детектором) – это атомарный элемент генотипа, в частности, хромосомы.

**Хромосома** представляет собой набор генов. Например, хромосому можно представить двоичной строкой, в которой каждый бит соответствует одному гену:

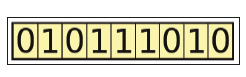
****

Рисунок Пример кодирования хромосомы

**Генотип** – набор генов, сгруппированных в хромосомы. В случае генетических алгоритмов каждому индивидууму соответствует хромосома.

В любой момент времени генетический алгоритм хранит **популяцию индивидуумов** – набор потенциальных решений поставленной задачи. Поскольку каждый индивидуум представлен некоторой хромосомой, эту популяцию можно рассматривать как коллекцию хромосом. Популяция всегда представляет текущее поколение и эволюционирует со временем, когда текущее поколение заменяется новым.

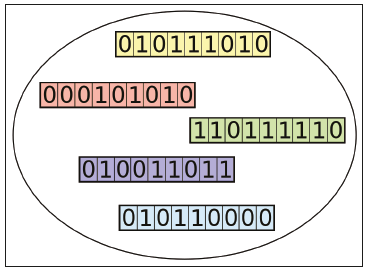
****

Рисунок Пример популяции

**Функция приспособленности**

На каждой итерации алгоритма индивидуумы оцениваются с помощью функции приспособленности (или целевой функции). Это функция, которую мы стремимся оптимизировать, или задача, которую пытаемся решить.

Индивидуумы, для которых функция приспособленности дает наилучшую оценку, представляют лучшие решения и с большей вероятностью будут отобраны для воспроизводства и представлены в следующем поколении. Со временем качество решений повышается, значения функции приспособленности растут, а когда будет найдено удовлетворительное значение, процесс можно остановить.

**Отбор**

После того как вычислены приспособленности всех индивидуумов в популяции, начинается процесс отбора, который определяет, какие индивидуумы будут оставлены для воспроизводства, т. е. создания потомков, образующих следующее поколение.

Процесс отбора основан на оценке приспособленности индивидуумов. Те,

чья оценка выше, имеют больше шансов передать свой генетический материал следующему поколению. Плохо приспособленные индивидуумы все равно могут быть отобраны, но с меньшей вероятностью. Таким образом, их генетический материал не полностью исключен.

**Скрещивание**

Для создания пары новых индивидуумов родители обычно выбираются из текущего поколения, а части их хромосом меняются местами (скрещиваются), в результате чего создаются две новые хромосомы, представляющие потомков. Эта операция называется скрещиванием, или рекомбинацией.

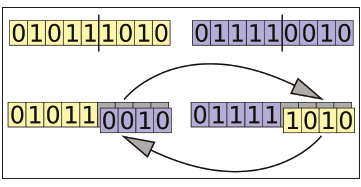
****

Рисунок Скрещивание с созданием двух потомков родителей

**Мутация:** Цель оператора мутации – периодически случайным образом обновлятьпопуляцию, т. е. вносить новые сочетания генов в хромосомы, стимулируя тем самым поиск в неисследованных областях пространства решений. Мутация может проявляться как случайное изменение гена. Мутации реализуются с помощью внесения случайных изменений в значения хромосом, например инвертирования одного бита в двоичной строке.

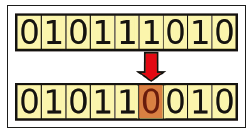
****

Рисунок Пример оператора мутации

## Отличия от традиционных алгоритмов

Между генетическими и традиционными алгоритмами поиска и оптимизации имеется несколько важных различий:

* **Поддержание популяции решений**

Целью генетического поиска является популяция потенциальных решений (индивидуумов), а не единственное решение. В любой точке поиска алгоритм сохраняет множество индивидуумов, образующих текущее поколение. На каждой итерации генетического алгоритма создается следующее поколение индивидуумов.

С другой стороны, в большинстве других алгоритмов поиска хранится единственное решение, которое итеративно улучшается. Например, алгоритм градиентного спускаитеративно сдвигает текущее решение в направлении наискорейшего спуска, которое определяется антиградиентом заданной функции.

* **Использование генетического представления решений**

Генетические алгоритмы работают не с самими потенциальными решениями, а с их кодированными представлениями, которые часто называют хромосомами. Простым примером хромосомы является двоичная строка фиксированной длины.

Хромосомы позволяют определить генетические операции скрещивания и мутации. Скрещивание реализуется обменом частей родительских хромосом, а мутация – изменением частей хромосом. Побочный эффект генетического представления – отделение поиска от исходной предметной области. Генетические алгоритмы не знают, что именно представляют хромосомы, и не пытаются их интерпретировать.

* **Использование функции приспособленности**

Функция приспособленности представляет проблему, которую мы пытаемся решить. Цель генетического алгоритма – найти индивидуумов, для которых оценка, вычисляемая функцией приспособленности, максимальна.

В отличие от традиционных алгоритмов поиска, генетические алгоритмы анализируют только значение, возвращенное функцией приспособленности, их не интересует ни производная, ни какая-либо другая информация. Поэтому они могут работать с функциями, которые трудно или невозможно продифференцировать.

* **Вероятностное поведение**

Многие традиционные алгоритмы по природе своей детерминированы, тогда как правила, применяемые генетическими алгоритмами для перехода от предыдущего поколения к следующему, вероятностные.

Например, вероятность отбора индивидуума для создания следующего поколения тем выше, чем больше значение функции приспособленности, но элемент случайности все равно присутствует. Слабо приспособленные индивидуумы могут быть отобраны, хотя вероятность этого ниже. Мутации тоже имеют вероятностный характер, обычно их вероятность мала, а изменению подвергаются случайные позиции в хромосоме. Случайность присутствует и в операторе скрещивания. В некоторых генетических алгоритмах скрещивание происходит лишь с некоторой вероятностью. Если скрещивания не было, то оба родителя дублируются в следующем поколении вообще без изменений.

Несмотря на вероятностную природу процесса, поиск, основанный на генетическом алгоритме, нельзя назвать случайным; случайность используется, чтобы направить поиск в сторону тех областей пространства поиска, где выше шансы улучшить результаты.

## Преимущества генетических алгоритмов

Особенности генетических алгоритмов определяют их преимущества по сравнению с традиционными алгоритмами поиска.

Преимущества генетических алгоритмов:

* способность выполнять глобальную оптимизацию;
* применимость к задачам со сложным математическим представлением;
* применимость к задачам, не имеющим математического представления;
* устойчивость к шуму;
* поддержка распараллеливания и распределенной обработки;
* пригодность к непрерывному обучению.

**Ограничения генетических алгоритмов**

* необходимы специальные определения;
* необходима настройка гиперпараметров;
* большой объем счетных операций;
* опасность преждевременной сходимости;
* отсутствие гарантированного решения.

## Базовая структура генетического алгоритма

На следующей блок-схеме показаны основные этапы генетического алгоритма.

****

Рисунок Этапы генетического алгоритма

## Применение генетических алгоритмов

Генетические алгоритмы применяются для решения следующих задач:

* Оптимизация функции
* Оптимизация запросов в базах данных
* Задачи на графах
* Настройка и обучение искусственной нейронной сети
* Задачи компоновки
* Игровые стратегии
* Теория приближений
* Биоинформатика

# Реализация модели

В качестве одного из возможных применений генетического алгоритма рассмотрим задачу коммивояжера.

Задача коммивояжера является одной из наиболее изученных задач оптимизации. Зачастую она используется для оценки алгоритмов оптимизации. У этой задачи много вариантов, но первоначально она ставилась на примере коммивояжера, которому требуется объехать несколько городов:

*Пусть дан список городов и известны расстояния между каждыми двумя городами. Найти кратчайший путь, проходящий через все города и возвращающийся в исходную точку.*

Количество возможных путей, проходящих через *n* городов, равно (*n* − 1)!/2. У каждого пути имеется стоимость – его длина, необходимо найти путь минимальной стоимости. В задаче коммивояжера города обычно представляются числами от 0 до *n* – 1, а возможные решения – последовательностями таких чисел. Например, в задаче с пятью городами решения имеют вид [0, 1, 2, 3, 4], [2, 4, 3, 1, 0] и т. д.

Каждое решение можно оценить, просуммировав расстояния между соседними в последовательности городами и прибавив еще расстояние между последним и первым городом. Такой список будем использовать для представления хромосомы в генетическом алгоритме.

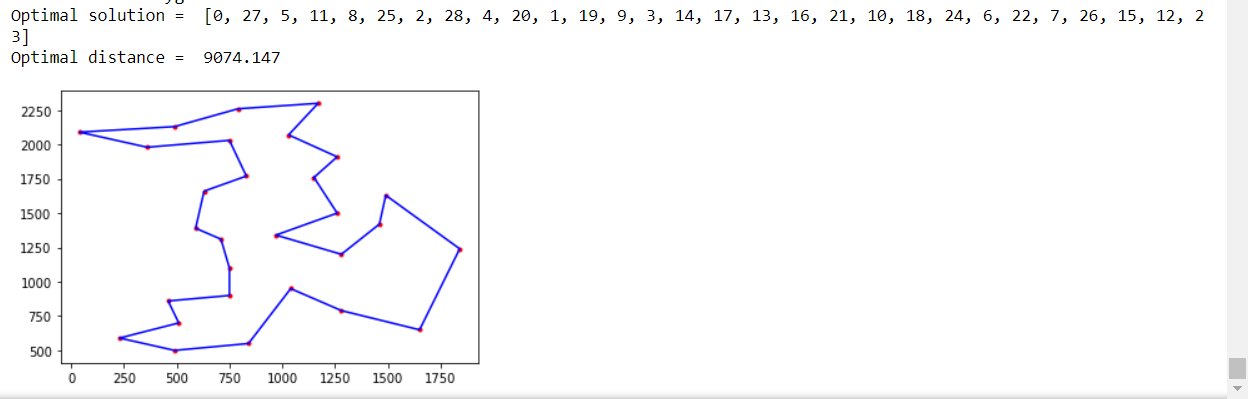
Для задачи коммивояжера был написан класс TravelingSalesmanProblem, находящийся в файле tsp.py. Этот класс предоставляет следующие закрытые методы:

* \_\_create\_data(): читает указанный файл из интернета, извлекает координаты городов, вычисляет попарные расстояния между городами и запоминает их в матрице расстояний (двумерном массиве). Затем сериализует местоположения городов и вычисленные расстояния в файле на диске с помощью встроенной утилиты pickle;
* \_\_read\_data(): читает сериализованные данные, а если их еще нет, вызывает \_\_create\_data() для их подготовки.

Кроме того, класс предоставляет следующие открытые методы:

* getTotalDistance(indices): вычисляет полную длину пути, проходящего через города с указанными индексами;
* plotData(indices): рисует путь, проходящий через города с указанными индексами.

При запуске данного класса как автономной Python-программы, будет выведено:



**Решение с помощью генетического алгоритма:**

1. Создаем экземпляр задачи



2. Затем нужно определить стратегию приспособления. В данной задаче требуется минимизировать состояние, а это значит, что нужно определить минимизирующий класс Fitness с одной целью, в котором задан один отрицательный вес:

 3. Показанное выше оптимальное решение задачи bayg29 представлено такой хромосомой: (0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23). Реализуем эту хромосому:

Сначала создается класс Individual, расширяющий массив целых чисел и дополняющий его атрибутом типа FitnessMin.



Затем регистрируется оператор randomOrder, который применяет функцию random.sample() к диапазону, соответствующему задаче коммивояжера (длины, равной количеству городов *n*). В результате генерируется случайный список индексов от 0 до *n* – 1.



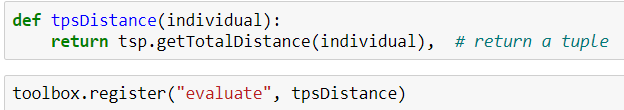
Далее регистрируется оператор IndividualCreator. Он вызывает оператор randomOrder и обходит созданный им список с целью создать хромосому, состоящую из индексов городов.



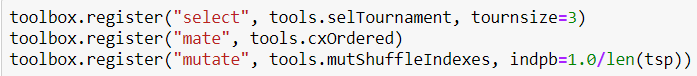
Также регистрируется оператор populationCreator, который порождает список индивидуумов, вызывая в цикле оператор IndividualCreator.



4. Определяем функцию вычисления приспособленности.

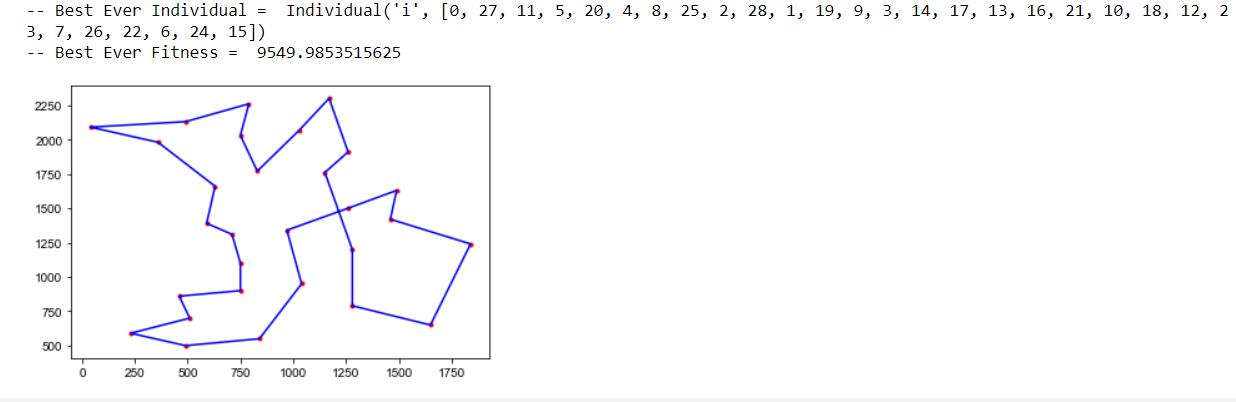


5. Определим генетических операторов. Будем использовать турнирный отбор с турниром размера 3, а также упорядоченное скрещивание

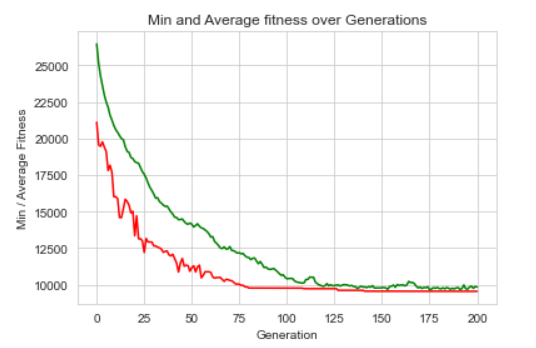


6. Реализация генетического алгоритма. Воспользуемся алгоритмом eaSimple (библиотека deap) и его объектами статистики, которые дают информацию, необходимую для последующего отображения результатов:

 Выполнение этой программы с заданными в начале файла константами (размер популяции 300, 200 поколений, вероятность скрещивания 0.9, вероятность мутации 0.1) дает следующие результаты:



На втором графике показана статистика работы генетического алгоритма.



# Список использованных источников

1. Д. Рутковская, М. Пилинський, Л. Рутковский. Нейронные сети, генетические алгоритмы и нечеткие системы. Горячая линия-Телеком. 2006 г. 193 стр.
2. Ротштейн А.П. Интеллектуальные технологии идентификации: нечеткие множества, нейронные сети, генетические алгоритмы. Монография. - Винница: "Універсум-Вінниця", 1999. - 295 с.
3. Eyal Wirsansky. Hands-On Genetic Algorithms with Python: Applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Packt Publishing. 2020. 346 p.

# Приложение

## 5.1. Код файла trav.ipynb

from deap import base

from deap import creator

from deap import tools

from deap import algorithms

import random

import array

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

import tsp

# установливаем случайное начальное число для повторяемых результатов

RANDOM\_SEED = 42

random.seed(RANDOM\_SEED)

# создаем экземпляр задачи коммивояжера

TSP\_NAME = "bayg29" # имя проблемы

tsp = tsp.TravelingSalesmanProblem(TSP\_NAME)

# Константы генетического алгоритма

POPULATION\_SIZE = 300

MAX\_GENERATIONS = 200

HALL\_OF\_FAME\_SIZE = 1

P\_CROSSOVER = 0.9 # вероятность кроссовера

P\_MUTATION = 0.1 # вероятность мутации особи

toolbox = base.Toolbox()

# опредяем единственную цель, сводя у минимуму фитнесс-стратегию

creator.create("FitnessMin", base.Fitness, weights=(-1.0,))

# создаем индивидуальный класс на основе списка целых чисел:

creator.create("Individual", array.array, typecode='i', fitness=creator.FitnessMin)

# создаем оператор, который генерирует случайным образом перемешанные индексы

toolbox.register("randomOrder", random.sample, range(len(tsp)), len(tsp))

# создаем индивидуальный оператор создания, чтобы заполнить Индивидуальный экземпляр перетасованными индексами

toolbox.register("individualCreator", tools.initIterate, creator.Individual, toolbox.randomOrder)

# создаем оператор создания популяции для создания списка людей

toolbox.register("populationCreator", tools.initRepeat, list, toolbox.individualCreator)

# расчет пригодности - вычислим общую дистанцию списка городов, представленных индексами

def tpsDistance(individual):

return tsp.getTotalDistance(individual), # return a tuple

toolbox.register("evaluate", tpsDistance)

# Генетические операторы

toolbox.register("select", tools.selTournament, tournsize=3)

toolbox.register("mate", tools.cxOrdered)

toolbox.register("mutate", tools.mutShuffleIndexes, indpb=1.0/len(tsp))

# Поток генетического алгоритма

def main():

# создаем начальную популяцию (поколение 0)

population = toolbox.populationCreator(n=POPULATION\_SIZE)

# подготовим объект статистики

stats = tools.Statistics(lambda ind: ind.fitness.values)

stats.register("min", np.min)

stats.register("avg", np.mean)

# определим объект HallOfFame

hof = tools.HallOfFame(HALL\_OF\_FAME\_SIZE)

# выполним поток генетического алгоритма с добавленной функцией hof

population, logbook = algorithms.eaSimple(population, toolbox, cxpb=P\_CROSSOVER, mutpb=P\_MUTATION,

ngen=MAX\_GENERATIONS, stats=stats, halloffame=hof, verbose=True)

# выведем лучшую индивидуальную информацию

best = hof.items[0]

print("-- Best Ever Individual = ", best)

print("-- Best Ever Fitness = ", best.fitness.values[0])

# график лучшего решения

plt.figure(1)

tsp.plotData(best)

# график статистики

minFitnessValues, meanFitnessValues = logbook.select("min", "avg")

plt.figure(2)

sns.set\_style("whitegrid")

plt.plot(minFitnessValues, color='red')

plt.plot(meanFitnessValues, color='green')

plt.xlabel('Generation')

plt.ylabel('Min / Average Fitness')

plt.title('Min and Average fitness over Generations')

# показ обоих графиков

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

## Код файла tsp.py

import csv

import pickle

import os

import codecs

import numpy as np

from urllib.request import urlopen

import matplotlib.pyplot as plt

class TravelingSalesmanProblem:

"""This class encapsulates the Traveling Salesman Problem.

City coordinates are read from an online file and distance matrix is calculated.

The data is serialized to disk.

The total distance can be calculated for a path represented by a list of city indices.

A plot can be created for a path represented by a list of city indices.

:param name: The name of the corresponding TSPLIB problem, e.g. 'burma14' or 'bayg29'.

"""

def \_\_init\_\_(self, name):

"""

Creates an instance of a TSP

:param name: name of the TSP problem

"""

# инициализируем переменные экземпляра

self.name = name

self.locations = []

self.distances = []

self.tspSize = 0

# инициализируем данные

self.\_\_initData()

def \_\_len\_\_(self):

"""

returns the length of the underlying TSP

:return: the length of the underlying TSP (number of cities)

"""

return self.tspSize

def \_\_initData(self):

"""Reads the serialized data, and if not available - calls \_\_create\_data() to prepare it

"""

# пытаемся прочитать сериализованные данные

try:

self.locations = pickle.load(open(os.path.join("tsp-data", self.name + "-loc.pickle"), "rb"))

self.distances = pickle.load(open(os.path.join("tsp-data", self.name + "-dist.pickle"), "rb"))

except (OSError, IOError):

pass

# если сериализованные данные не найдены - создаем данные с нуля:

if not self.locations or not self.distances:

self.\_\_createData()

# устанавливаем проблему 'размер':

self.tspSize = len(self.locations)

def \_\_createData(self):

"""Reads the desired TSP file from the Internet, extracts the city coordinates, calculates the distances

between every two cities and uses them to populate a distance matrix (two-dimensional array).

It then serializes the city locations and the calculated distances to disk using the pickle utility.

"""

self.locations = []

# открываем файл с разделителями-пробелами из url-адреса и читаем из него строки:

with urlopen("http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/" + self.name + ".tsp") as f:

reader = csv.reader(codecs.iterdecode(f, 'utf-8'), delimiter=" ", skipinitialspace=True)

# пропускаем строки, пока не будет найдена одна из этих строк:

for row in reader:

if row[0] in ('DISPLAY\_DATA\_SECTION', 'NODE\_COORD\_SECTION'):

break

# читаем строки данных до тех пор, пока не найдем EOF:

for row in reader:

if row[0] != 'EOF':

# удалим индекс в начале строки:

del row[0]

# преобразуем координаты x, y в ndarray:

self.locations.append(np.asarray(row, dtype=np.float32))

else:

break

# установим проблему 'размер':

self.tspSize = len(self.locations)

# выведем данные:

print("length = {}, locations = {}".format(self.tspSize, self.locations))

# инициализируем матрицу расстояний, заполнив ее нулями:

self.distances = [[0] \* self.tspSize for \_ in range(self.tspSize)]

# заполним матрицу расстояний вычисленными расстояниями:

for i in range(self.tspSize):

for j in range(i + 1, self.tspSize):

# вычислим евклидово расстояние между двумя ndarrays:

distance = np.linalg.norm(self.locations[j] - self.locations[i])

self.distances[i][j] = distance

self.distances[j][i] = distance

print("{}, {}: location1 = {}, location2 = {} => distance = {}".format(i, j, self.locations[i], self.locations[j], distance))

# сериализируем местоположения и расстояния:

if not os.path.exists("tsp-data"):

os.makedirs("tsp-data")

pickle.dump(self.locations, open(os.path.join("tsp-data", self.name + "-loc.pickle"), "wb"))

pickle.dump(self.distances, open(os.path.join("tsp-data", self.name + "-dist.pickle"), "wb"))

def getTotalDistance(self, indices):

"""Calculates the total distance of the path described by the given indices of the cities

:param indices: A list of ordered city indices describing the given path.

:return: total distance of the path described by the given indices

"""

# расстояние между последним и первым городом:

distance = self.distances[indices[-1]][indices[0]]

# сложим расстояние между каждой парой следующих друг за другом городов:

for i in range(len(indices) - 1):

distance += self.distances[indices[i]][indices[i + 1]]

return distance

def plotData(self, indices):

"""plots the path described by the given indices of the cities

:param indices: A list of ordered city indices describing the given path.

:return: the resulting plot

"""

# нарисуем точки, представляющие города:

plt.scatter(\*zip(\*self.locations), marker='.', color='red')

# создадим список соответствующих населенных пунктов города:

locs = [self.locations[i] for i in indices]

locs.append(locs[0])

# проведем линию между каждой парой последовательных городов:

plt.plot(\*zip(\*locs), linestyle='-', color='blue')

return plt

# тестирование

def main():

# создадим экземпляр проблемы:

tsp = TravelingSalesmanProblem("bayg29")

optimalSolution = [0, 27, 5, 11, 8, 25, 2, 28, 4, 20, 1, 19, 9, 3, 14, 17, 13, 16, 21, 10, 18, 24, 6, 22, 7, 26, 15, 12, 23]

print("Problem name: " + tsp.name)

print("Optimal solution = ", optimalSolution)

print("Optimal distance = ", tsp.getTotalDistance(optimalSolution))

# построим решение:

plot = tsp.plotData(optimalSolution)

plot.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()