**Лабораторная работа**

**Метаэвристические решения задачи коммивояжёра.**

В данной работе рассматриваем и реализуем на Python различные подходы (GA, ACO, SA) к решению задачи коммивояжёра.

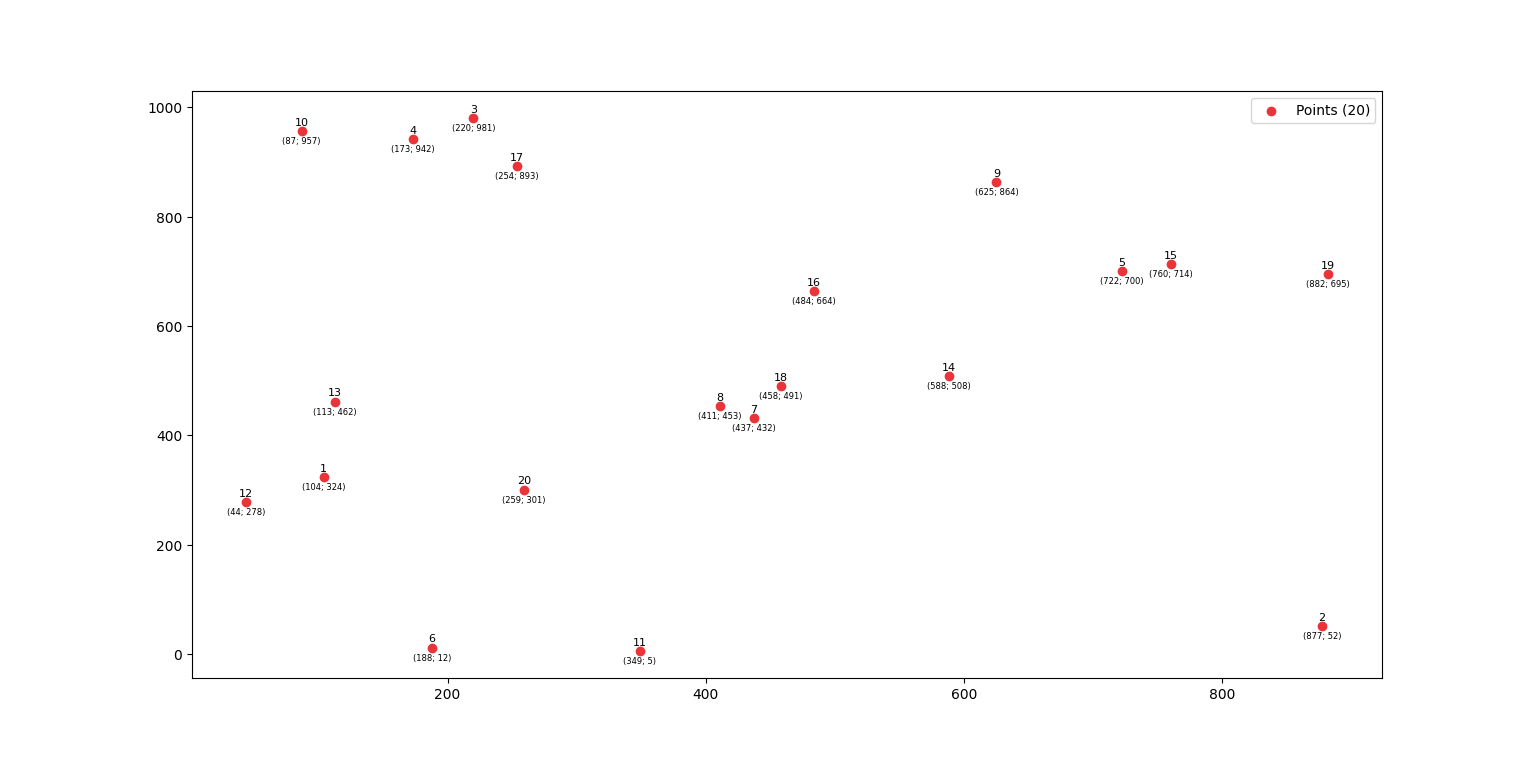
**Постановка задачи.**

Разработать программу решения **задачи коммивояжера**. Найти оптимальный маршрут движения коммивояжера между городами, проходящий через все промежуточные пункты по одному разу и возвращающийся в исходную точку исключающий повторное посещение городов. Известны координаты городов (расстояния между городами).

**Решение.**

Для возможности применения математического аппарата к решению задачи мы представим её в виде симметричного графа. Вершины графа соответствуют условным городам, а рёбра — путям сообщения между этими городами. Каждому ребру сопоставим критерий выгодности маршрута — расстояние между городами. В целях упрощения задачи будем считать, что граф является полностью связным.

Для визуализации задачи будем использовать класс TSP (программный код представлен ниже). Этот класс включает в себя функции, необходимые для отрисовки точек, маршрутов, а также интерактивной легенды, на элементы которой можно нажимать для того, чтобы скрыть / отобразить выбранные пути. Сгенерируем набор точек, воспользовавшись функцией points = generate\_problem(20), и выведем его на экран с помощью TSP(points=points, paths=None).



Файл TSP.py

# Travelling Salesman Problem

from random import randint

from numpy import array

from matplotlib import pyplot as plt

from matplotlib.lines import Line2D

from matplotlib.backend\_bases import PickEvent

from algorithms.utils.path import Path

def generate\_problem(count: int, canvas\_size: int = 1000) -> list[tuple[int]]:

"""Generates a list of random 2D points."""

return [(randint(0, canvas\_size), randint(0, canvas\_size)) for \_ in range(count)]

class TSP:

"""

Allows to visualize the Traveling Salesman Problem and paths.

"""

CLR\_POINT = "#eb343a"

CLR\_PATH = [

"#eb343a",

"#db34eb",

"#5b34eb",

"#34b4eb",

"#34eb4c",

"#ebe534",

"#eb9234",

]

def \_\_init\_\_(self, points: list[tuple[int]], paths: list[Path] = None) -> None:

"""Initializes the problem, outputs its data using graphics."""

self.\_\_points = points

self.\_\_paths = paths

self.\_\_fig, self.\_\_ax = plt.subplots(num=f"Travelling Salesman Problem")

self.\_\_show()

def get\_points(self) -> list[tuple[int]]:

"""Getter to get the list of 2D points of the initialized problem."""

return self.\_\_points

def get\_paths(self) -> list[Path]:

"""Getter to get the list of paths of the initialized problem."""

return self.\_\_paths

def \_\_draw\_points(self) -> None:

"""Draws 2D points and their coordinates on the canvas."""

self.\_\_ax.scatter(

\*array(self.\_\_points).T,

zorder=1,

color=TSP.CLR\_POINT,

label=f"Points ({len(self.\_\_points)})",

)

for i, p in enumerate(self.\_\_points):

plt.annotate(

i + 1,

p,

ha="center",

textcoords="offset points",

xytext=(0, 4),

fontsize=8,

)

plt.annotate(

f"({p[0]}; {p[1]})",

p,

ha="center",

va="top",

textcoords="offset points",

xytext=(0, -4),

fontsize=6,

)

def \_\_draw\_paths(self) -> list[Line2D]:

"""Draws all given paths on the canvas."""

lines = []

if self.\_\_paths:

for i, path in enumerate(self.\_\_paths):

points = [self.\_\_points[i] for i in path.indx]

(l,) = plt.plot(

\*array(points).T,

ls="--",

zorder=0,

color=TSP.CLR\_PATH[i % len(TSP.CLR\_PATH)],

label=f"{path.name} ({path.leng:.2f})",

)

lines.append(l)

return lines

def \_\_draw\_legend(self, lines: list[Line2D]) -> None:

"""Draws the legend on the canvas."""

if lines:

self.\_\_ax.set\_title(

"Tip: Click on the legend line(s) to turn the path ON / OFF",

fontsize=10,

loc="left",

)

legend = self.\_\_ax.legend()

lined = {}

for legline, origline in zip(legend.get\_lines(), lines):

legline.set\_picker(True)

lined[legline] = origline

def on\_pick(event: PickEvent) -> None:

legline = event.artist

origline = lined[legline]

visible = not origline.get\_visible()

origline.set\_visible(visible)

legline.set\_alpha(1.0 if visible else 0.2)

self.\_\_fig.canvas.draw()

self.\_\_fig.canvas.mpl\_connect("pick\_event", on\_pick)

else:

self.\_\_ax.legend()

def \_\_show(self) -> None:

"""Shows the canvas with the drawn data."""

self.\_\_draw\_points()

lines = self.\_\_draw\_paths()

self.\_\_draw\_legend(lines=lines)

plt.show()

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

pass

Стоит отметить, что в программной реализации решениями является список объектов класса Path, которые содержат в себе информацию о последовательности точек маршрута, о его длине и названии (используются для легенды на графике).

Файл Path.py

# Path

from dataclasses import dataclass

@dataclass

class Path:

"""

Dataclass describing a path using:

\* list of point indices;

\* path length;

\* path name (optional).

"""

indx: list[int]

leng: float

name: str

Но откуда берутся эти решения?

Поскольку коммивояжёр в каждом из условных городов встаёт перед выбором следующего города из тех, что он ещё не посетил, существует (n-1)! / 2 маршрутов для симметричной задачи. Если городов, например, 20, то возможных маршрутов будет 60 822 550 204 416 000. Для того, чтобы перебрать такое количество вариантов и найти оптимальный путь, понадобятся огромные вычислительные мощности и куча времени.

К счастью, на практике зачастую не требуется находить именно точное решение. Достаточно и субоптимального, которое будет близко к оптимальному пути.

При такой постановке задачи для её решения можно воспользоваться метаэвристическими алгоритмами, которые значительно ускорят процесс нахождения решения.

**Метаэвристические алгоритмы**

Метаэвриистика — метод оптимизации, многократно использующий простые правила для достижения субоптимального решения.

В данной работе мы рассмотрим следующие алгоритмы решения задачи:

1. **Генетический алгоритм** (Genetic Algorithm, GA), относящийся к классу эволюционных методов;
2. **Муравьиный алгоритм** (Ant Colony Optimization, ACO), относящийся к классу методов “роевого” интеллекта;
3. **Алгоритм имитации отжига** (Simulated annealing, SA), относящийся к классу методов, имитирующих физические процессы.

Каждый из вышеописанных алгоритмов будет использовать методы для вычислений расстояний между точками. Объединим данный функционал в класс Base, от которого в дальнейшем будем наследовать общие методы.

Файл Base.py

# Base

from math import sqrt

class Base:

"""

The base class for path finding algorithms.

Contains common functions.

"""

@staticmethod

def \_\_euclidean\_dist(a: tuple[int], b: tuple[int]) -> float:

"""Calculates the Euclidean distance between two 2D points."""

return sqrt((a[0] - b[0]) \*\* 2 + (a[1] - b[1]) \*\* 2)

@staticmethod

def \_calculate\_dist(dm: list[list[float]], indx: list[int]) -> float:

"""Calculates the path length based on the index list of the distance matrix."""

dist = 0

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

dist += dm[indx[i]][indx[i + 1]]

return dist

@staticmethod

def \_distance\_matrix(points: list[tuple[int]]) -> list[list[float]]:

"""Calculates the distance matrix for the given 2D points."""

return [[Base.\_\_euclidean\_dist(a, b) for b in points] for a in points]

Здесь:

* euclidean\_dist() используется для нахождения Евклидова расстояния между двумя 2D точками;
* calculate\_dist() рассчитывает длину указанного пути на основе матрицы расстояний;
* distance\_matrix(), собственно, эту матрицу вычисляет.

**Генетический алгоритм**

Генетический алгоритм — алгоритм поиска, используемый для решения задач оптимизации и моделирования путём случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе.

Данные механизмы применяются к особям, каждая из которых в программной реализации представляет из себя список индексов 2D точек — последовательность, в которой эти точки следует обойти для достижения кратчайшего пути. Начальная популяция состоит из особей, случайно сгенерированных функцией initialization().

После создания начальной популяции запускается цикл операций, направленных на постепенную эволюцию особей:

* fitness\_sort() ранжирует особи в текущей популяции с помощью функции приспособления (чем короче маршрут, содержащийся в геноме особи, тем лучше);
* selection() отбирает часть наиболее приспособленных особей из текущей популяции;
* crossover() скрещивает особей текущей популяции, соединяя части генома от двух случайно выбранных родителей (панмиксия);
* mutation() случайным образом делает перестановки в геноме некоторых особей текущей популяции.

Когда цикл завершит свою работу, мы используем лучшую особь (маршрут) из последней популяции в качестве решения задачи.

Отметим, что результаты работы алгоритма во многом зависят от заданных гиперпараметров:

* population — общее число особей, участвующих в одной итерации;
* iter — максимальное количество итераций алгоритма;
* s — определяет, сколько лучших особей попадет в следующую популяцию (в процентном соотношении);
* m — определяет, как часто особи в популяции мутируют (в процентном соотношении).

Ниже приводится программная реализация данного алгоритма.

Файл GA.py

# Genetic Algorithm

from random import random, shuffle, sample

from .utils.base import Base

from .utils.path import Path

class GA(Base):

"""

Genetic algorithm is a metaheuristic inspired by the process of natural selection

that belongs to the larger class of evolutionary algorithms.\n

-----

`population: int` THE NUMBER OF INDIVIDUALS\n

The total number of individuals involved in one iteration.\n

-----

`iter: int` THE NUMBER OF ITERATIONS\n

The maximum number of iterations of the algorithm.\n

-----

`s: float` SELECTION COEFFICIENT\n

Determines how many of the best individuals will make it to the next population.\n

-----

`m: float` MUTATION COEFFICIENT\n

Determines how often individuals in a population mutate.\n

"""

def \_\_init\_\_(self, population: int, iter: int, s: float, m: float) -> None:

"""Initializes the hyperparameters for the algorithm."""

self.population = population

self.iter = iter

self.s = s

self.m = m

@staticmethod

def \_\_fitness\_sort(dm: list[list[float]], individuals: list[list[int]]) -> None:

"""Sorts the individuals of a given population by fit."""

individuals.sort(key=lambda i: GA.\_calculate\_dist(dm, i))

def \_\_initialization(self, l: int) -> list[list[int]]:

"""Initializes the first population of individuals."""

base = list(range(l))

individuals = []

for \_ in range(self.population):

shuffle(base)

individuals.append(base + [base[0]])

return individuals

def \_\_selection(self, individuals: list[list[int]]) -> None:

"""Selects the best individuals of a given population."""

del individuals[int(self.population \* self.s) :]

def \_\_crossover(self, individuals: list[list[int]]) -> None:

"""Crossbreeding some individuals of a given population."""

childs = []

w\_size = len(individuals[0]) // 2

for \_ in range(len(individuals), self.population):

p1, p2 = sample(individuals, 2)

childs.append(

p1[: w\_size - 1]

+ [i for i in p2[:-1] if i not in p1[: w\_size - 1]]

+ [p1[0]]

)

individuals += childs

def \_\_mutation(self, individuals: list[list[int]]) -> None:

"""Mutates some individuals of a given population."""

sampling = list(range(1, len(individuals[0]) - 1))

for item in individuals:

if random() < self.m:

i, j = sample(sampling, 2)

item[i], item[j] = item[j], item[i]

def run(self, points: list[tuple[int]], name: str = None) -> Path:

"""Runs the algorithm for the given 2D points."""

l = len(points)

dm = GA.\_distance\_matrix(points)

individuals = self.\_\_initialization(l)

for \_ in range(self.iter):

GA.\_\_fitness\_sort(dm, individuals)

self.\_\_selection(individuals)

self.\_\_crossover(individuals)

self.\_\_mutation(individuals)

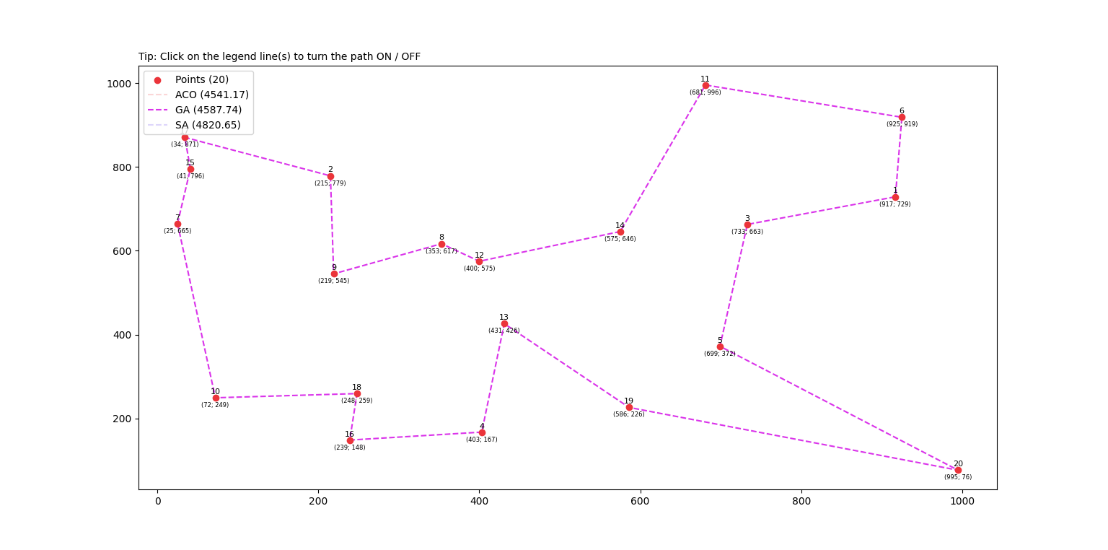
GA.\_\_fitness\_sort(dm, individuals)

return Path(indx=individuals[0], leng=GA.\_calculate\_dist(dm, individuals[0]), name=name)

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

pass

Такой результат получается при запуске алгоритма с набором гиперпараметров GA(population=1500, iter=40, s=0.2, m=0.5):



**Муравьиный алгоритм**

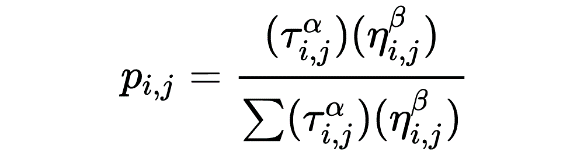
Муравьиный алгоритм — один из эффективных алгоритмов для нахождения приближённых решений задачи коммивояжёра, а также решения аналогичных задач поиска маршрутов на графах. Суть подхода заключается в анализе и использовании модели поведения муравьёв, ищущих пути от колонии к источнику питания.

Изначально условные муравьи передвигаются случайным образом, охватывая различные маршруты. Во время передвижения муравьи оставляют след из феромона, обозначая эти самые маршруты. Соответственно, чем короче получился путь, тем большее количество феромона за одинаковое количество времени муравьи на нём оставят. Феромон с течением времени испаряется. Из-за этого на длинных маршрутах остаётся меньше феромона, чем на коротких, даже если по ним передвигалось одинаковое количество муравьёв. Муравьи будут чаще выбирать самую “пахучую тропинку” и забывать пути с меньшим количеством феромона.

В программной реализации мы на каждой итерации цикла имитируем поведение каждого муравья из колонии:

* create\_indx() формирует маршрут муравья (список индексов точек) и подсчитывает длину;
* select\_i() отвечает за выбор муравьём каждой последующей точки маршрута внутри функции create\_indx();
* update\_pm() обновляет матрицу феромона.

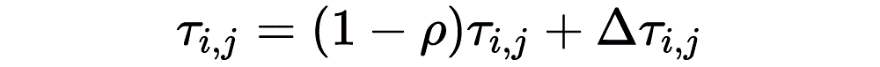
Формулой ниже описывается вероятность, с которой муравей перейдёт из точки i в точку j (используется в функции create\_indx()):



Здесь:

* τ(i,j) — количество феромона на ребре i, j (значение из матрицы феромона);
* α — гиперпараметр, контролирующий влияние τ(i,j) (задаётся пользователем);
* η(i,j) — привлекательность ребра i, j, равняется 1 / d(i,j) (обратному значению из матрицы расстояний);
* β — гиперпараметр, контролирующий влияние η(i,j) (задаётся пользователем).

После формирования маршрутов необходимо обновить матрицу феромона (за это отвечает функция update\_pm()). Обновление происходит по следующему правилу:



Здесь:

* τ(i,j) — количество феромона на ребре i, j (значение из матрицы феромона);
* ρ — скорость испарения феромона (задаётся пользователем);
* Δτ(i,j) — количество отложенного феромона, равняется q / d(i,j) (интенсивность феромона, делённая на длину маршрута из точки i в точку j).

В конце каждой итерации находим лучший маршрут из маршрутов, пройденных на текущей итерации, и сравниваем его с абсолютно лучшим найденным маршрутом. Если новое решение оказалось лучше, то записываем его в отдельную переменную. Последний записанный маршрут и будет являться решением задачи.

Так же, как и генетический алгоритм, данный алгоритм зависим от гиперпараметров:

* ants — общее количество агентов (муравьев), задействованных в одной итерации;
* iter — максимальное количество итераций алгоритма;
* α — коэффициент, контролирующий влияние феромона на ребре;
* β —коэффициент, контролирующий влияние привлекательности маршрута (величина, обратно пропорциональная длине маршрута);
* ρ — коэффициент испарения феромона, который отражает степень взаимного влияния между муравьями, как правило, значение равно [0, 1], что предотвращает бесконечное накопление феромона;
* q — интенсивность феромона, которая представляет собой общее количество феромонов, в определённой степени влияет на скорость сходимости алгоритма.

Ниже приводится программная реализация данного алгоритма.

Файл ACO.py

# Ant Colony Optimization

from math import inf

from random import random, shuffle

from .utils.base import Base

from .utils.path import Path

class ACO(Base):

"""

Ant Colony Optimization algorithm is introduced based on the foraging behavior of an ant

for seeking a path between their colony and source food.\n

-----

`ants: int` THE NUMBER OF ANTS\n

The total number of agents (ants) involved in one iteration.\n

-----

`iter: int` THE NUMBER OF ITERATIONS\n

The maximum number of iterations of the algorithm.\n

-----

`a: float` INFORMATION ELICITATION FACTOR\n

The information elicitation factor α, which represents the relative importance of the pheromone,

reflects the importance of the accumulation of the pheromone with regard to the ants' path selection.\n

-----

`b: float` EXPECTED HEURISTIC FACTOR\n

The expected heuristic factor β, which represents the relative importance of the visibility,

reflects the importance of the heuristic information with regard to the ants' path selection.\n

-----

`p: float` PHEROMONE EVAPORATION COEFFICIENT\n

The pheromone evaporation coefficient ρ, which represents the degree of pheromone evaporation,

reflects the degree of mutual influence among ants. Generally, the value of is [0, 1],

which prevents the infinite accumulation of pheromone effectively.\n

-----

`q: float` PHEROMONE INTENSITY\n

The pheromone intensity Q, which represents the total pheromone,

affects the convergence speed of the alghoritm to a certain extent.\n

"""

def \_\_init\_\_(self, ants: int, iter: int, a: float, b: float, p: float, q: float) -> None:

"""Initializes the hyperparameters for the algorithm."""

self.ants = ants

self.iter = iter

self.a = a

self.b = b

self.p = p

self.q = q

@staticmethod

def \_\_select\_i(selection: list[int]) -> int:

"""Selects a random index of the next 2D point."""

sum\_num = sum(selection)

if sum\_num == 0:

return len(selection) - 1

tmp\_num = random()

prob = 0

for i in range(len(selection)):

prob += selection[i] / sum\_num

if prob >= tmp\_num:

return i

def \_\_create\_indx(self, dm: list[list[float]], pm: list[list[float]]) -> list[int]:

"""Creates a new ordering of 2D point indices based on the distance and pheromone."""

l = len(dm)

unvisited\_indx = list(range(l))

shuffle(unvisited\_indx)

visited\_indx = [unvisited\_indx.pop()]

for \_ in range(l - 1):

i = visited\_indx[-1]

selection = []

for j in unvisited\_indx:

selection.append(

(pm[i][j] \*\* self.a) \* ((1 / max(dm[i][j], 10\*\*-5)) \*\* self.b)

)

selected\_i = ACO.\_\_select\_i(selection)

visited\_indx.append(unvisited\_indx.pop(selected\_i))

visited\_indx.append(visited\_indx[0])

return visited\_indx

def update\_pm(self, pm: list[list[float]], tmp\_indx: list[list[int]], tmp\_leng: list[float]) -> None:

"""Updates the pheromone matrix."""

l = len(pm)

for i in range(l):

for j in range(i, l):

pm[i][j] \*= 1 - self.p

pm[j][i] \*= 1 - self.p

for i in range(self.ants):

delta = self.q / tmp\_leng[i]

indx = tmp\_indx[i]

for j in range(l):

pm[indx[j]][indx[j + 1]] += delta

pm[indx[j + 1]][indx[j]] += delta

def run(self, points: list[tuple[int]], name: str = None) -> Path:

"""Runs the algorithm for the given 2D points."""

l = len(points)

dm = ACO.\_distance\_matrix(points)

pm = [[1 for \_ in range(l)] for \_ in range(l)]

res\_indx = []

res\_leng = inf

for \_ in range(self.iter):

tmp\_indx = []

tmp\_leng = []

for \_ in range(self.ants):

indx = self.\_\_create\_indx(dm, pm)

tmp\_indx.append(indx)

tmp\_leng.append(ACO.\_calculate\_dist(dm, indx))

self.update\_pm(pm, tmp\_indx, tmp\_leng)

best\_leng = min(tmp\_leng)

if best\_leng < res\_leng:

res\_leng = best\_leng

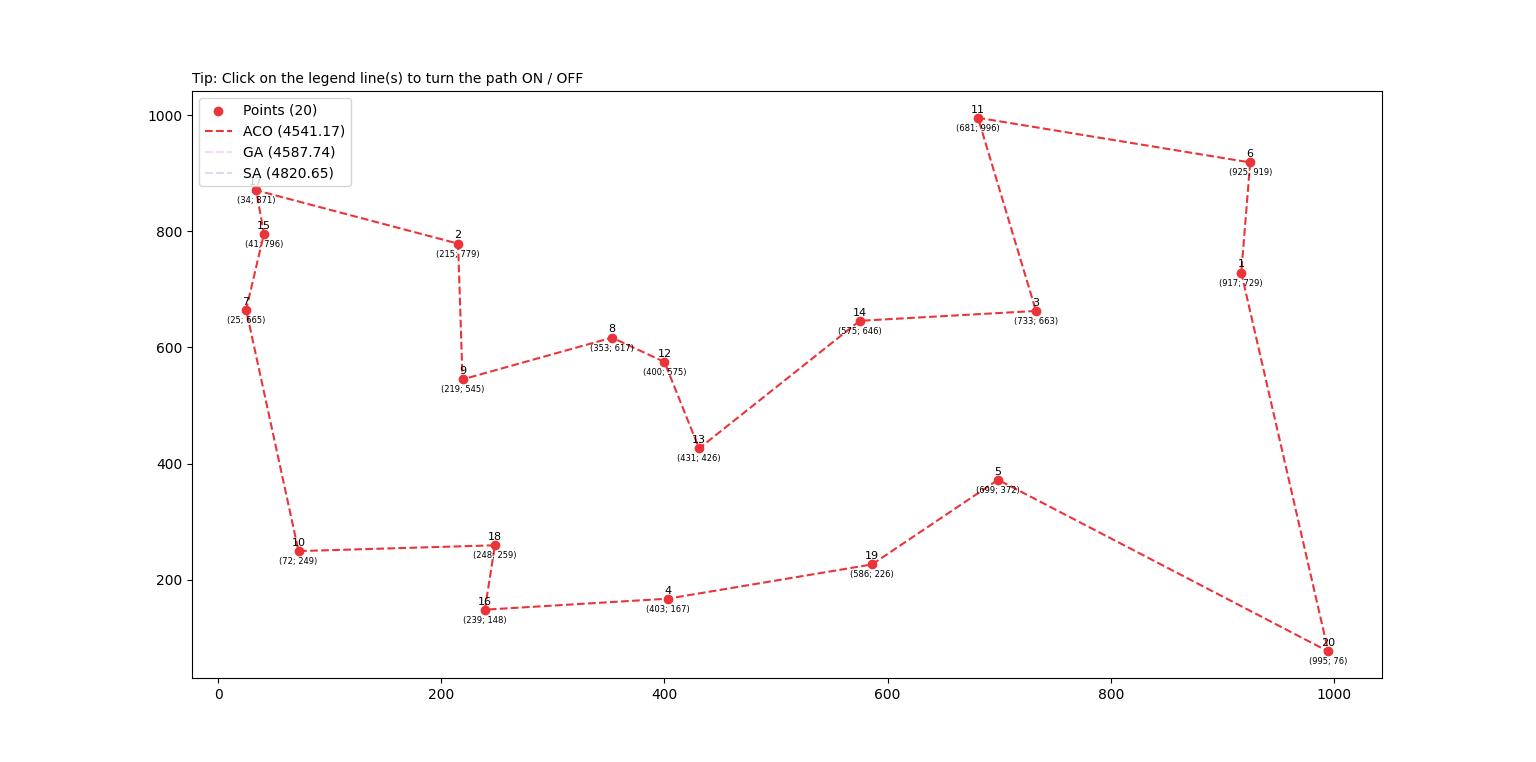
res\_indx = tmp\_indx[tmp\_leng.index(best\_leng)]

return Path(indx=res\_indx, leng=res\_leng, name=name)

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

pass

Такой результат получается при запуске алгоритма с набором гиперпараметров ACO(ants=100, iter=20, a=1.5, b=1.2, p=0.6, q=10):



**Алгоритм имитации отжига**

Алгоритм имитации отжига — метод решения задачи глобальной оптимизации. Он основывается на имитации физического процесса, который происходит при кристаллизации вещества, в том числе при отжиге металлов. Предполагается, что в определённый момент атомы вещества практически выстроены в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Активность атомов тем больше, чем выше температура, которую постепенно понижают, что приводит к тому, что вероятность переходов в состояния с большей энергией уменьшается. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте.

В программной реализации под энергией понимается длина получившегося маршрута. Изначально маршрут задаётся случайно. На каждой итерации с помощью перестановки пары индексов мы генерируем новый маршрут. На основе его длины, длины предыдущего маршрута и текущего значения температуры функция is\_acceptable() определяет, совершать ли переход в новое состояние или нет. Вероятность перехода рассчитывается по формуле:

Метаэвристические решения Задачи Коммивояжёра 6

Здесь:

* ΔF(i) — изменение “энергии” (разность между длинами вероятного и текущего маршрута);
* T(i) — значение температуры на i-ой итерации алгоритма.

В конце каждой итерации система “остывает”: параметр, отвечающий за температуру, уменьшается. Когда алгоритм отработает отведённое количество итераций, он вернёт минимальный найденный маршрут, являющийся решением задачи.

Гиперпараметры алгоритма:

* iter — максимальное количество итераций алгоритма;
* t — начальная температура поиска, уменьшается по мере продвижения поиска;
* g — коэффициент, влияющий на изменение температуры.

Ниже приводится программная реализация данного алгоритма.

Файл SA.py

# Simulated Annealing

# from math import exp

from numpy import exp

from random import sample, random

from .utils.base import Base

from .utils.path import Path

class SA(Base):

"""

Simulated annealing is a probabilistic technique for approximating the global optimum of a given function.

Specifically, it is a metaheuristic to approximate global optimization in a large search space for an optimization problem.\n

-----

`iter: int` THE NUMBER OF ITERATIONS\n

The maximum number of iterations of the algorithm.\n

-----

`t: int` INITIAL TEMPERATURE\n

The initial temperature for the search decreases with the progress of the search.\n

-----

`g: float` CHANGE COEFFICIENT\n

The coefficient affecting temperature change.\n

"""

def \_\_init\_\_(self, iter: int, t: int, g: float) -> None:

"""Initializes the hyperparameters for the algorithm."""

self.iter = iter

self.t = t

self.g = g

def \_\_is\_acceptable(self, prb\_leng: float, tmp\_leng: float) -> bool:

"""Checks if the state transition will execute."""

prob = min(1, exp(-(prb\_leng - tmp\_leng) / self.t))

if prob > random():

return True

return False

def run(self, points: list[tuple[int]], name: str = None) -> Path:

"""Runs the algorithm for the given 2D points."""

l = len(points)

dm = SA.\_distance\_matrix(points)

tmp\_indx = [i for i in range(l)] + [0]

tmp\_leng = SA.\_calculate\_dist(dm, tmp\_indx)

res\_indx = tmp\_indx.copy()

res\_leng = tmp\_leng

for \_ in range(self.iter):

i, j = sample(range(1, l), 2)

prb\_indx = tmp\_indx.copy()

prb\_indx[i], prb\_indx[j] = prb\_indx[j], prb\_indx[i]

prb\_leng = SA.\_calculate\_dist(dm, prb\_indx)

if self.\_\_is\_acceptable(prb\_leng, tmp\_leng):

tmp\_indx = prb\_indx

tmp\_leng = prb\_leng

if tmp\_leng < res\_leng:

res\_indx = tmp\_indx

res\_leng = tmp\_leng

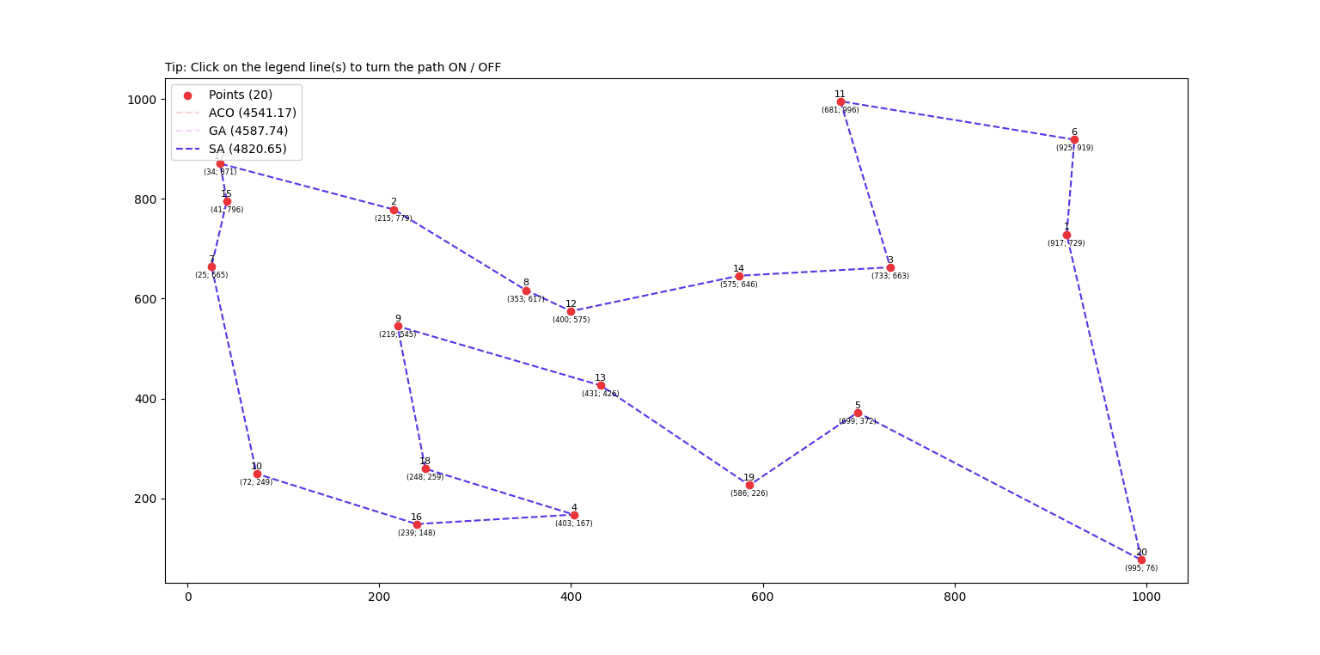
self.t \*= self.g

return Path(indx=res\_indx, leng=res\_leng, name=name)

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

pass

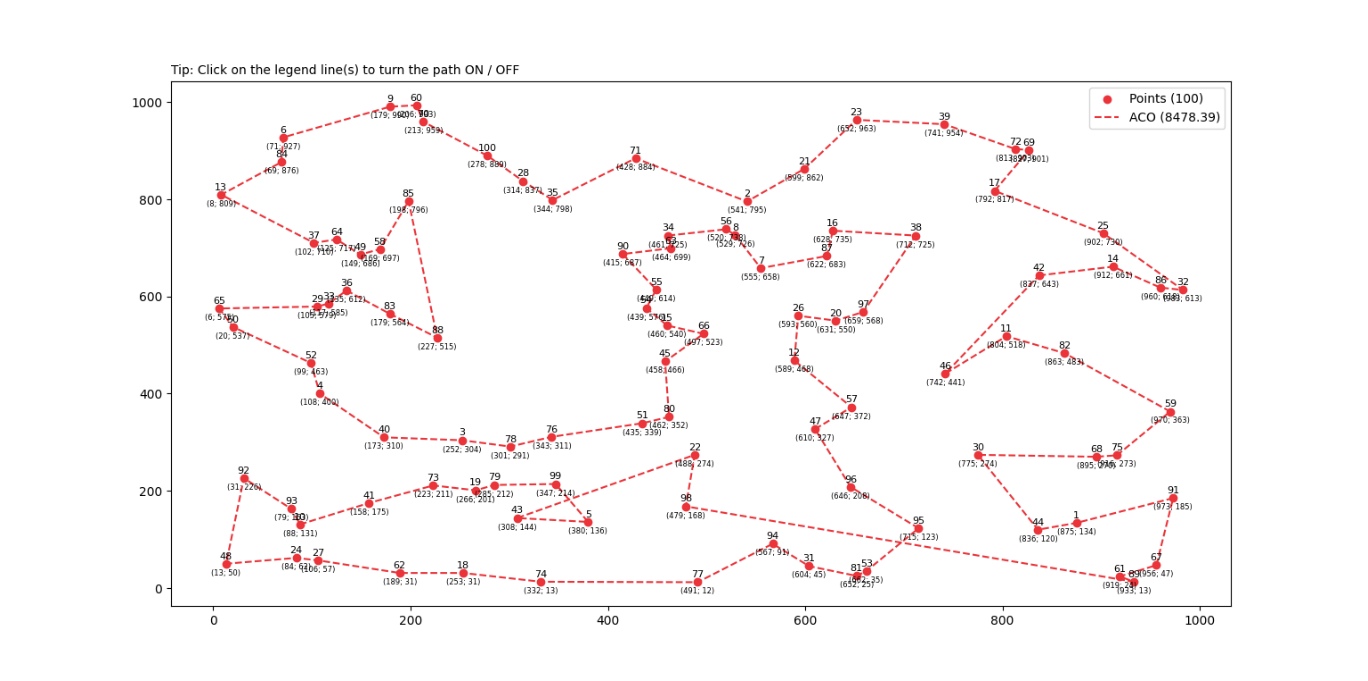
Такой результат получается при запуске алгоритма с набором гиперпараметров SA(iter=20000, t=100, g=0.6):

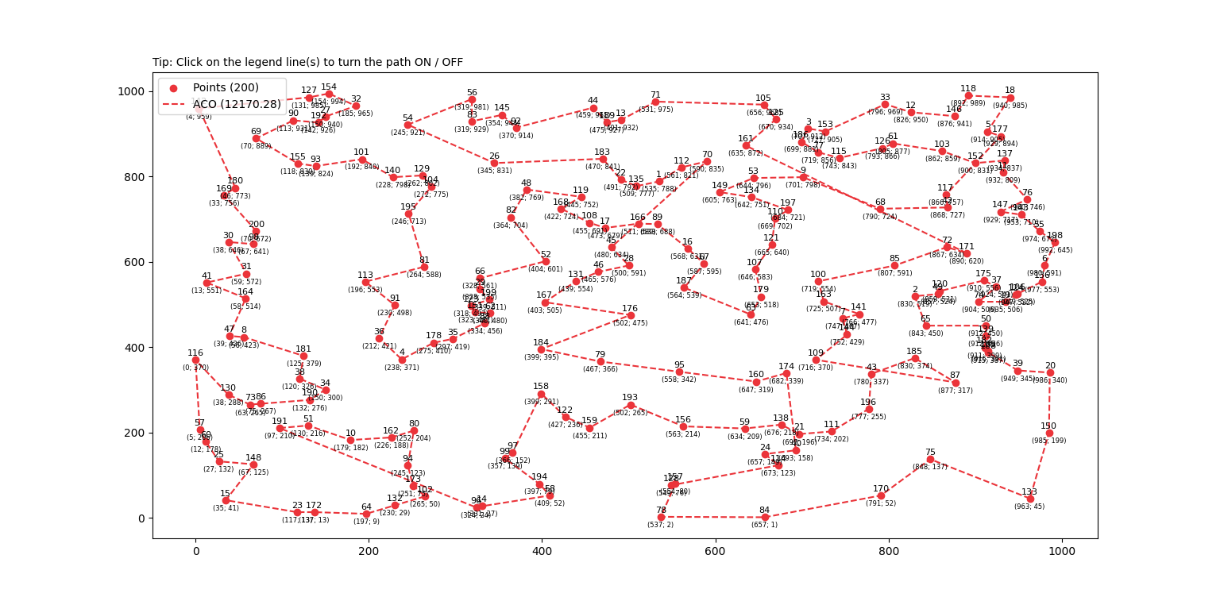


**Сравнение алгоритмов**

Сравнивать алгоритмы затруднительно, потому что — как упоминалось выше — результаты во многом зависят от подобранных гиперпараметров. Так же нужно учитывать, что существует довольно много различных вариаций реализации алгоритмов, в зависимости от которых результаты тоже будут отличаться.

Если же выбирать лучшую реализацию из тех трёх, что были описаны в данной статье, то я бы отдал предпочтение муравьиному алгоритму. Он довольно быстр и, что наиболее важно, показывает хорошие результаты даже при большом количестве исходных данных. Ниже приводятся примеры решения задачи на 100 и 200 точек.





**Заключение**

Таким образом, мы разобрали одни из самых популярных метаэвристических подходов к решению задачи коммивояжёра: генетический алгоритм, муравьиный алгоритм и алгоритм иммитации отжига.

**Задания для самостоятельного выполнения.**

Вариант задания определяется по номеру фамилии в списке группы либо по согласованию с преподавателем.

Допускается реализация задания на любом из изученных языков программирования.

1. Разработать программу решения производственной задачи. Цех может производить стулья и столы. На производство стула идет 5 единиц материала, на производство стола – 20 единиц. Для изготовления стула требуется 10 человеко- часов, стола – 15 Имеется 400 единиц материала и 450 человеко-часов. Прибыль при производстве стула – 45 у.е., при производстве стола – 80 у.е. Сколько надо сделать стульев и столов, чтобы получить максимальную прибыль?

2. Разработать программу решения задачи раскроя. Найти оптимальный вариант раскроя некоторого количества рулонов бумаги фиксированной ширины для различных заказчиков (которым нужны различные количества рулонов различной ширины), минимизировав при этом отходы.

3. Разработать программу решения транспортной задачи. Составить оптимальный план перевозок между N складами и K магазинами, при котором стоимость перевозок будет минимальна. Известна потребность в товаре каждым магазином, наличие товара на складах и стоимость перевозки единицы продукции с каждого склада до каждого магазина.

4. Разработать программу поиска минимального остовного дерева. Есть несколько городов, которые необходимо соединить дорогами так, чтобы можно было добраться из любого города в любой другой (напрямую или через другие города). Разрешается строить дороги между заданными парами городов, и известна стоимость строительства каждой такой дороги. Требуется решить, какие именно дороги нужно строить, чтобы минимизировать общую стоимость строительства.

5. Разработать программу решения задачи о максимальном потоке. Как (т.е. по каким маршрутам) послать максимально возможное количество грузов из начального пункта в конечный пункт, если пропускная способность путей между пунктами ограничена?

6. Разработать программу решения задачи о назначениях. Имеется некоторое число работ и некоторое число исполнителей. Любой исполнитель может быть назначен на выполнение любой (но только одной) работы, но с неодинаковыми затратами. Нужно распределить работы так, чтобы выполнить их с минимальными затратами.

7. Разработать программу о назначении целей. Найти оптимальное распределение комплекта различного вооружения для поражения целей для нанесения максимального поражения противнику.

8. Разработать программу решения задачи о загрузке (ранце).

Из заданного множества предметов со свойствами «стоимость» и «вес» требуется отобрать некое число предметов таким образом, чтобы получить максимальную суммарную стоимость при одновременном соблюдении ограничения на суммарный вес.

9. Разработать программу решения задачи о кратчайшем пути. Как кратчайшим путем (с наименьшим расходом топлива и времени, т.е. дешевле) попасть из пункта А в пункт Б?

10. Разработать программу размещения локальной сети. Найти оптимальную конфигурацию прокладки сетевого кабеля и коммутационного оборудования.