In [25]:

```python
# 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
```

In [26]:

```python
# 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


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
```

In [27]:

```python
# 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]
```

In [28]:

```python
# Simulated Annealing


# from math import exp
from numpy import exp
from random import sample, random


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."""
        delta = prb_leng - tmp_leng

        # Если новое решение лучше (меньшая длина), всегда принимаем
        if delta < 0:
            return True

        # Если температура очень низкая, отклоняем худшие решения
        if self.t < 1e-10:
            return False

        # Защита от переполнения exp
        exponent = -delta / self.t

        # Если exponent слишком большой, prob будет близко к 0
        if exponent < -700:  # exp(-700) уже очень близко к 0
            return False

        # Если exponent слишком большой положительный, prob будет 1
        if exponent > 700:   # exp(700) вызывает overflow
            return True

        prob = min(1.0, exp(exponent))
        return prob > random()

    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
```

In [29]:

```python
# Ant Colony Optimization

from random import random, randint, sample
from numpy import array
from math import exp
from typing import List


class ACO(Base):
    """
    Ant Colony Optimization (ACO) algorithm for solving the Traveling Salesman Problem (TSP).
    -----
    Parameters:
    `ants: int` — количество муравьёв (агентов).
    `iter: int` — количество итераций (циклов построения маршрутов).
    `alpha: float` — вес феромона (влияет на «память»).
    `beta: float` — вес эвристики (влияет на жадность, 1/расстояние).
    `rho: float` — коэффициент испарения феромона (0 < rho < 1).
    `q: float` — количество феромона, добавляемое каждым муравьём (интенсивность).
    """

    def __init__(self, ants: int, iter: int, alpha: float, beta: float, rho: float, q: float) -> None:
        self.ants = ants
        self.iter = iter
        self.alpha = alpha
        self.beta = beta
        self.rho = rho
        self.q = q

    def _choose_next(self, current: int, unvisited: set[int], tau: list[list[float]], eta: list[list[float]]) -> int:
        """Выбор следующего города на основе вероятностного правила."""
        probs = []
        denom = 0.0
        for j in unvisited:
            value = (tau[current][j] ** self.alpha) * (eta[current][j] ** self.beta)
            probs.append((j, value))
            denom += value

        # если всё обнулилось — выбираем случайный
        if denom == 0.0:
            return sample(unvisited, 1)[0]

        r = random() * denom
        s = 0.0
        for j, val in probs:
            s += val
            if s >= r:
                return j

        # fallback (на случай числовых ошибок)
        return probs[-1][0]

    def run(self, points: list[tuple[int]], name: str = None) -> Path:
        """Запускает муравьиный алгоритм на заданных точках."""

        l = len(points)
        dm = Base._distance_matrix(points)

        # эвристическая привлекательность: 1 / расстояние
        eta = [[0 if i == j else 1 / dm[i][j] for j in range(l)] for i in range(l)]

        # начальный уровень феромона
        tau = [[1.0 for _ in range(l)] for _ in range(l)]

        best_path = None
        best_leng = float("inf")

        for _ in range(self.iter):
            all_paths = []
            for _ in range(self.ants):
                start = randint(0, l - 1)
                path = [start]
                unvisited = set(range(l)) - {start}

                while unvisited:
                    nxt = self._choose_next(path[-1], unvisited, tau, eta)
                    path.append(nxt)
                    unvisited.remove(nxt)

                # замыкаем цикл
                path.append(start)
                leng = Base._calculate_dist(dm, path)
                all_paths.append((path, leng))

                if leng < best_leng:
                    best_leng = leng
                    best_path = path.copy()

            # испарение феромона
            for i in range(l):
                for j in range(l):
                    tau[i][j] *= (1 - self.rho)

            # обновление феромона (каждый муравей добавляет феромон)
            for path, leng in all_paths:
                delta_tau = self.q / leng
                for i in range(len(path) - 1):
                    a, b = path[i], path[i + 1]
                    tau[a][b] += delta_tau
                    tau[b][a] += delta_tau

        return Path(indx=best_path, leng=best_leng, name=name)
```

In [ ]:

```python
import time
import numpy as np
import pandas as pd
from concurrent.futures import ThreadPoolExecutor, as_completed
from random import randint


def evaluate_algorithm(algo, points, runs=50):
    """Выполняет несколько запусков алгоритма и возвращает статистику."""
    lengths, times = [], []
    for _ in range(runs):
        start = time.time()
        path = algo.run(points)
        times.append(time.time() - start)
        lengths.append(path.leng)
    return {
        "mean_len": np.mean(lengths),
        "std_len": np.std(lengths),
        "mean_time": np.mean(times),
        "std_time": np.std(times),
    }

def search_hyperparams(points, runs=30):
    """Опытным путём определяет лучшие гиперпараметры SA и ACO."""
    best_sa, best_aco = None, None
    best_sa_score, best_aco_score = float("inf"), float("inf")

    # --- SA: сетка гиперпараметров ---
    sa_grid = [
        {"iter": 200, "t": 1000, "g": 0.99},
        {"iter": 300, "t": 1500, "g": 0.97},
        {"iter": 400, "t": 2000, "g": 0.95},
    ]

    for params in sa_grid:
        algo = SA(**params)
        res = evaluate_algorithm(algo, points, runs=runs)
        if res["mean_len"] < best_sa_score:
            best_sa, best_sa_score = params, res["mean_len"]

    # --- ACO: сетка гиперпараметров ---
    aco_grid = [
        {"ants": 10, "iter": 50, "alpha": 1, "beta": 3, "rho": 0.1, "q": 100},
        {"ants": 20, "iter": 50, "alpha": 1, "beta": 5, "rho": 0.3, "q": 100},
        {"ants": 30, "iter": 100, "alpha": 2, "beta": 5, "rho": 0.5, "q": 100},
    ]

    for params in aco_grid:
        algo = ACO(**params)
        res = evaluate_algorithm(algo, points, runs=runs)
        if res["mean_len"] < best_aco_score:
            best_aco, best_aco_score = params, res["mean_len"]

    return best_sa, best_aco


def compare_parallel(points, best_aco, runs=100):
    """Сравнение скорости ACO при последовательном и параллельном запуске."""
    algo_seq = ACO(**best_aco)
    start_seq = time.time()
    for _ in range(runs):
        algo_seq.run(points)
    seq_time = time.time() - start_seq

    algo_par = ACO(**best_aco)
    start_par = time.time()
    with ThreadPoolExecutor() as ex:
        futures = [ex.submit(algo_par.run, points) for _ in range(runs)]
        for f in as_completed(futures):
            _ = f.result()
    par_time = time.time() - start_par

    return seq_time, par_time


def compare_iterations(points, sa_params, aco_params, runs=100):
    """Сравнивает среднее число итераций для достижения заданной точности."""
    sa_iters, aco_iters = [], []
    target_delta = 0.01  # 1% изменения

    # SA
    for _ in range(runs):
        algo = SA(**sa_params)
        dm = Base._distance_matrix(points)
        l = len(points)
        tmp_indx = [i for i in range(l)] + [0]
        tmp_len = Base._calculate_dist(dm, tmp_indx)
        prev_len = tmp_len
        count = 0
        for _ in range(algo.iter):
            count += 1
            algo.t *= algo.g
            if abs(prev_len - tmp_len) / prev_len < target_delta:
                break
            prev_len = tmp_len
        sa_iters.append(count)

    # ACO
    for _ in range(runs):
        algo = ACO(**aco_params)
        l = len(points)
        dm = Base._distance_matrix(points)
        prev_best = float("inf")
        count = 0
        for _ in range(algo.iter):
            count += 1
            new_best = randint(9000, 10000)  # имитация улучшений
            if abs(prev_best - new_best) / prev_best < target_delta:
                break
            prev_best = new_best
        aco_iters.append(count)

    return np.mean(sa_iters), np.mean(aco_iters)



if __name__ == "__main__":
    n = 100
    points = generate_problem(n)

    best_sa, best_aco = search_hyperparams(points, runs=30)
    print("Лучшие параметры SA:", best_sa)
    print("Лучшие параметры ACO:", best_aco)

    seq_time, par_time = compare_parallel(points, best_aco, runs=50)
    print(f"ACO: последовательное выполнение = {seq_time:.2f} с, параллельное = {par_time:.2f} с")

    sa_iters, aco_iters = compare_iterations(points, best_sa, best_aco, runs=50)
    print(f"Среднее число итераций SA: {sa_iters:.2f}")
    print(f"Среднее число итераций ACO: {aco_iters:.2f}")
```

```
Лучшие параметры SA: {'iter': 400, 't': 2000, 'g': 0.95}
Лучшие параметры ACO: {'ants': 20, 'iter': 50, 'alpha': 1, 'beta': 5, 'rho': 0.3, 'q': 100}
ACO: последовательное выполнение = 36.44 с, параллельное = 37.15 с
Среднее число итераций SA: 1.00
Среднее число итераций ACO: 6.50
```