МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт компьютерных наук и кибербезопасности

Высшая школа технологий искусственного интеллекта

Отчёт по дисциплине «Генетические алгоритмы»

Лабораторная работа №3

«Решение задачи коммивояжера с помощью генетических алгоритмов»

Вариант №17

Студент: _____ Салимли Айзек Мухтар Оглы

Преподаватель: _____ Большаков Александр Афанасьевич

«____»_____ 20__ г.

Санкт-Петербург, 2025

Содержание

Введение

Задача коммивояжера (Traveling Salesman Problem, TSP) является одной из самых известных и изучаемых проблем комбинаторной оптимизации. Её классическая формулировка заключается в следующем: коммивояжеру необходимо посетить ровно по одному разу каждый из заданного множества городов и вернуться в исходный пункт. Расстояния между всеми парами городов известны. Цель состоит в том, чтобы найти такой порядок обхода городов (замкнутый маршрут или гамильтонов цикл), при котором общее пройденное расстояние является минимально возможным, а единственным ограничением является требование посетить каждый город ровно один раз перед возвращением. Математически задача может быть представлена на взвешенном графе $G = (V, E)$, где $V = \{1, 2, \ldots, n\}$— множество вершин (городов), а E — множество рёбер (путей между городами). Каждому ребру (i, j) приписан неотрицательный вес d_{ij} , представляющий расстояние или стоимость перемещения из города i в город j. Задача сводится к нахождению минимального по весу гамильтонова цикла в этом графе. Пусть бинарная переменная x_{ij} принимает значение 1, если ребро (i, j) включено в маршрут, и 0 в противном случае. Тогда целевая функция, которую необходимо минимизировать, записывается как:

$$\sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} d_{ij} x_{ij}$$

Для того чтобы решение образовывало единственный замкнутый цикл, а не набор несвязных подциклов, на переменные x_{ij} накладываются следующие ограничения. Во-первых, гарантируется, что коммивояжер въезжает в каждый город ровно один раз и выезжает из него ровно один раз:

$$\sum_{i=1, i \neq j}^{n} x_{ij} = 1, \ \ \forall j, \quad \sum_{j=1, j \neq i}^{n} x_{ij} = 1 \ \ \forall i$$

Во-вторых, вводятся так называемые "субтурные"или "подцикловые"ограничения, которые исключают формирование циклов, не включающих все города:

$$\sum_{i \in S} \sum_{j \notin S} x_{ij} \geq 1, \ \ \forall S \subset V, \ \ S \neq \emptyset$$

Именно наличие этих экспоненциально растущих ограничений делает задачу коммивояжера NP-трудной,тоестьдляеёточногорешениянесуществуетизвестного эффективного (полиномиального) алгоритма. В связи с этим для решения практических задач крупного масштаба широко применяются метаэвристические подходы, среди которых особое место занимают генетические алгоритмы, позволяющие находить близкие к оптимальным решения за разумное время путём имитации процессов естественной эволюции.

1 Постановка задачи

Дано: Набор данных wi29 (29 городов в Западной Сахаре): Матрица расстояний получается путем нахождения эвклидовых расстояний между координатами города по формуле:

$$Dist = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Требуется:

1. Реализовать с использованием генетических алгоритмов решение задачи коммивояжера;
2. Представить графически найденное решение и график сходимости алгоритма;
3. Проанализировать ключевые параметры алгоритма и их влияние на поиск решения;
4. Описать используемые подходы в реализации, представление особей и операторы генетического алгоритма.

Ограничения: Эвклидовы координаты городов.

2 Представление хромосом в задаче коммивояжера

2.1 Представление соседства

В представлении соседства тур кодируется вектором $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, где v_i обозначает город, следующий за городом i. Например, вектор $(2, 4, 8, 3, 9, 7, 1, 5, 6)$ представляет тур $1 \to 2 \to 4 \to 3 \to 8 \to 5 \to 9 \to 6 \to 7 \to 1$. Каждый корректный тур имеет единственное представление, однако не каждый вектор соответствует допустимому решению — возможны частичные циклы, как в $(2, 4, 8, 1, 9, 3, 5, 7, 6)$, где возникает подцикл $1 \to 2 \to 4 \to 1$.

Для данного представления определены три специализированных оператора кроссовера:

1. *Alternating edges*: последовательное чередование рёбер от родителей. Если добавляемое ребро образует цикл, выбирается случайное допустимое ребро из того же родителя.
2. *Subtour chunks*: выбор случайного подтура от одного родителя, затем фрагмента от другого. При возникновении циклов применяется репарация.
3. *Heuristic crossover*: начинается со случайного города, на каждом шаге выбирается кратчайшее ребро из двух родительских вариантов. При зацикливании добавляется случайный непосещённый город.

Математически, эвристический кроссовер минимизирует локальные приращения длины:

$$v_i = \arg \min_{j \in \{p_i, q_i\}} d_{ij},$$

где p_i, q_i — соседи города i у родителей.

Преимущество представления — естественные строительные блоки (рёбра), описываемые схемами вида $(* * * 3 \ * \ 7 \ * * *)$ для рёбер $(4, 3)$ и $(6, 7)$. Однако операторы *alternating edges* и *subtour chunks* проявляют высокую разрушительность, тогда как *heuristic crossover* демонстрирует наилучшую, но нестабильную производительность за счёт учёта метрики d_{ij}.

2.2 Порядковое представление

В порядковом представлении тур кодируется вектором $\mathbf{o} = (o_1, o_2, \ldots, o_n)$, где $o_i \in \{1, 2, \ldots, n - i + 1\}$. Данное представление основано на фиксированном упорядоченном списке городов-ориентиров $\mathbf{C} = (c_1, c_2, \ldots, c_n)$.

Процесс декодирования осуществляется последовательным выбором городов из $\mathbf{C}$ согласно индексам o_i:

$$\begin{aligned} t_1 &= C[o_1], \\ t_2 &= \big(C \setminus \{t_1\}\big)[o_2], \\ t_3 &= \big(C \setminus \{t_1, t_2\}\big)[o_3], \\ &\vdots \\ t_i &= \big(C \setminus \{t_1, \ldots, t_{i-1}\}\big)[o_i]. \end{aligned}$$

Ключевое преимущество представления — корректность классического одноточечного кроссовера. Для родителей $\mathbf{o}^p = (o_1^p, \ldots, o_n^p)$ и $\mathbf{o}^q = (o_1^q, \ldots, o_n^q)$ потомки получаются разрезом в позиции k:

$$\begin{aligned} \mathbf{o}^{\text{child1}} &= \big(o_1^p, \ldots, o_k^p,\ o_{k+1}^q, \ldots, o_n^q\big), \\ \mathbf{o}^{\text{child2}} &= \big(o_1^q, \ldots, o_k^q,\ o_{k+1}^p, \ldots, o_n^p\big). \end{aligned}$$

Математически гарантируется, что $\forall i :\ o_i^{\text{child}} \in \{1, \ldots, n - i + 1\}$, что обеспечивает корректность результирующих туров. Однако левые части $(i \le k)$ сохраняются нетронутыми, а правые $(i > k)$

подвергаются случайной перестановке, что приводит к слабым эволюционным характеристикам и ограниченной практической применимости.

2.3 Путевое представление

Путевое представление является, возможно, наиболее естественным представлением тура. Например, тур $5-1-7-8-9-4-6-2-3$ представляется просто как последовательность городов: $(5,1,7,8,9,4,6,2,3)$.

Для этого типа представления широко известны три операции кроссовера: частично отображённый (partially-mapped, PMX), порядковый (order, OX) и циклический (cycle, CX) кроссоверы.

- **PMX (partially-mapped crossover)** строит потомков, выбирая подпоследовательность из тура одного из родителей и сохраняя порядок и последовательность наибольшего возможного числа городов другого родителя. Подпоследовательность выбирается двумя случайными точками разреза.

  Например, для родителей с разметкой разрезов:

  $$\mathbf{p}_1 = (1,2,3 \mid 4,5,6,7 \mid 8,9),$$
  $$\mathbf{p}_2 = (4,5,2 \mid 1,8,7,6 \mid 9,3),$$

  сегменты между точками разреза меняются местами, что приводит к временному состоянию потомков:

  $$\mathbf{o}_1 = (X,X,X \mid 4,5,6,7 \mid X,X), \quad \mathbf{o}_2 = (X,X,X \mid 1,8,7,6 \mid X,X),$$

  где символы X обозначают неизвестные элементы. Затем с помощью серии преобразований данных:

  $$1 \leftrightarrow 4, \quad 8 \leftrightarrow 5, \quad 7 \leftrightarrow 6,$$

  оставшиеся позиции заполняются значениями из другого родителя с разрешением конфликтов, в результате чего получаются корректные потомки.

- **OX (order crossover)** строит потомков, копируя сегмент между двумя точками из одного родителя, а оставшиеся города заполняются в порядке появления в другом родителе, начиная с точки сразу после второго разреза.

  При тех же родителях и точках разреза:

  $$\mathbf{p}_1 = (1,2,3 \mid 4,5,6,7 \mid 8,9), \quad \mathbf{p}_2 = (4,5,2 \mid 1,8,7,6 \mid 9,3),$$

  потомки могут выглядеть так:

  $$\mathbf{o}_1 = (X,X,X \mid 4,5,6,7 \mid X,X), \quad \mathbf{o}_2 = (X,X,X \mid 1,8,7,6 \mid X,X),$$

  после заполнения оставшихся городов из другого родителя по порядку, начиная после второго разреза:

  $$\mathbf{o}_1 = (2,1,8 \mid 4,5,6,7 \mid 9,3), \quad \mathbf{o}_2 = (3,4,5 \mid 1,8,7,6 \mid 9,2).$$

- **CX (cycle crossover)** строит потомков, где каждый город и его позиция приходят от одного из родителей. Начинаем с первого города первого родителя и находим его соответствие во втором родителе, затем продолжаем циклически, возвращаясь к первому городу.

  Например, для родителей

  $$\mathbf{p}_1 = (1,2,3,4,5,6,7,8,9), \quad \mathbf{p}_2 = (4,5,2,1,8,7,6,9,3),$$

первый потомок начинается как

$$\mathbf{o}_1 = (1, X, X, X, X, X, X, X),$$

затем заполняется городами по циклу: $4, 8, 3, 2$, после чего оставшиеся города берутся из второго родителя, что приводит к:

$$\mathbf{o}_1 = (1, 2, 3, 4, 7, 6, 9, 8, 5).$$

Аналогично формируется второй потомок, начиная с первого города второго родителя.

В представленной реализации используется оператор PMX.

3 Программная реализация

В рамках лабораторной работы №2, реализован генетический алгоритм (ГА) для **задачи коммивояжёра (TSP)**. Используются евклидовы расстояния, предвычисленные в матрицу D по заданным координатам 29 городов.

3.1 Структура и настройки

- **Данные и константы:**
  - `RAW_COORDS` — список (ID, x, y) для 29 городов.
  - `CITY_IDS` — отображаемые номера городов на графиках.
  - `COORDS` $\in \mathbb{R}^{N\times 2}$ — массив координат.
  - `DIST = build_distance_matrix(COORDS)` — полная матрица попарных расстояний.
- **Конфигурация ГА** (`GAConfig`; значения в `main()`):
  - `pop_size` $= 220$, `max_gens` $= 2000$, `seed` $= 42$.
  - `p_cx` $= 0.7$ (кроссовер OX), `p_mut` $= 0.05$, `elitism` $= 1$.
  - Отбор: турнир `tour_k` $= 2$.
  - Диверсификация: иммиграция каждые `immigrants_period` $= 25$ поколений.
  - Fitness sharing для отбора: `sharing_alpha` $= 0.12$, `sharing_sample` $= 40$.
  - Адаптивный стоп-критерий: окно `stop_window` $= 60$, порог относительного улучшения `stop_rel_improve` $= 0.003$, порог разрыва `stop_gap_rel` $= 0.003$
  - `hard_patience` $= 80$ поколений без улучшения лучшего — принудительная остановка.
- **Сетка параметров (для CSV-сводок):**
  - Метки кроссовера: `p_c_labels = ["0.5", "0.7", "0.9"]`.
  - Метки мутации: `p_m_labels = ["0.02", "0.05", "0.10"]`.
  - Повторов на конфигурацию: `runs_per_conf = 30`.
  - База зерна: `seed_base = 123456` (для устойчивого рандомизации по конфигурациям).

3.2 Модель данных и история

- **Маршрут/индивид:** перестановка $\pi \in S_N$ (массив индексов городов длиной N).
- **Целевая функция (длина тура):**

$$L(\pi) \;=\; \sum_{i=1}^{N} D_{\pi_i,\,\pi_{i+1}}, \quad \pi_{N+1} \equiv \pi_1.$$

- **История прогона** (возвращаемая `evolve_tsp`): лучший маршрут, лучшая длина, `history_best` (лучшее по поколениям), `history_mean` (среднее по поколениям), `snapshots` (лучшие на контрольных поколениях), `last_generation`, `runtime_sec`.

Инициализация и оценка

- `init_population(n, pop_size, rng)` — равномерная случайная инициализация перестановок (`rng.permutation`).
- `lengths(pop, D)` — векторизованный расчёт длин туров для популяции (циклический сдвиг `np.roll`).
- `build_distance_matrix(coords)` — попарные евклидовы расстояния (вектор `diff`, `np.hypot`).

Селекция

- `tournament_selection(pop, fit_for_selection, k, rng)` — турнирный выбор победителей по *штрафованным* длинам; реализует мягкое давление ($k = 2$).
- `sharing_penalty(pop, base_lengths, rng, alpha, sample_size)` — fitness sharing: множитель $1 + \alpha \cdot \text{similarity}$ по среднему сходству (Хэмминг по позициям) к случайному поднабору популяции; *меняет только* значения для отбора, не реальную длину.

Кроссовер (OX)

- `ox_crossover(p1, p2, rng)` — классический *Order Crossover*: случайный сегмент копируется из $p1$, оставшиеся гены дозаполняются в порядке обхода $p2$ с обходом циклом; корректно обрабатывает вырожденный выбор границ (соседние точки).
- Применение: попарно по родителям с вероятностью `p_cx`; иначе — копирование.

Мутация

- `mutate_inversion(ind, rng)` — инверсия подотрезка $[i, j]$, при $\Pr = 0.1$ — дополнительная редкая перестановка пары генов (swap).
- Применяется к каждому потомку независимо с вероятностью `p_mut`.

Элитизм и иммиграция

- Элитизм `elitism`: сохраняются elitism лучших из старого поколения и *переписывают* столько же худших в новом.
- Иммигранты: каждые `immigrants_period` поколений доля `immigrants_ratio` худших заменяется *новыми случайными* перестановками; после этого *проверка стоп-критерия* пропускается на `stop_suspend_after_imm` поколений.

Основной цикл

- `evolve_tsp(D, cfg)` — повторяет: *элитизм* → *fitness sharing* → *турнир* → *OX-кроссовер* → *мутация* → *элитизм возврат* → *иммиграция (периодически)* → *оценка* → *логирование метрик и снимков*, применяет адаптивный стоп-критерий и «жёсткую» страховку.
- Контрольные поколения (*чекпоинты*): $\{0, 10, 20, 50, 80, 100, 120, 150, 180\}$; на них сохраняется лучший маршрут в `snapshots`.

3.3 Визуализация

- `plot_route(coords, route, title, filename)` — отрисовка тура по координатам с подписями `CITY_IDS`; сохраняет PNG (`route_gen_*.png`, `best_route.png`).
- `plot_convergence(best, mean_vals, checkpoints, filename)` — график сходимости: лучшая и средняя длина по поколениям; чекпоинты отмечаются маркерами; сохраняется как `convergence.png`.

3.4 Сводки и CSV (свип параметров)

- `run_grid_and_save_csv(D, base_cfg, p_c_labels, p_m_labels, runs_per_conf, ...)` — перебор конфигураций (P_c, P_m) с множественными запусками; для каждой конфигурации логируются лучшая длина, число поколений, время выполнения (мс).
- **Выходные CSV:**
  - `runs_log.csv` — подробный лог всех прогонов (seed, метки P_c, P_m, численные значения, лучшая длина, поколения, время).
  - `summary_distance.csv` — матрица средних длин с стандартным отклонением по узлам $(P_c \times P_m)$: ячейки вида `mean` $\pm$ `std`.
  - `summary_runtime_ms.csv` — матрица средних времен (в мс) по $(P_c \times P_m)$.
- Для репликации стохастики по конфигурациям используется `seed_base` в сочетании с хэшем `(pc_label, pm_label, run_index)`.

3.5 Сценарии запуска (entry points)

1. `main()`:
   - Запускает `evolve_tsp` с базовой конфигурацией; печатает лучшую длину, номер последнего поколения и время.
   - Сохраняет `convergence.png`, `route_gen_*.png` для достигнутых чекпоинтов и `best_route.png`.
   - Сохраняет краткую сводку одиночного прогона в `single_run_summary.csv`.
   - Запускает свип по (P_c, P_m) и формирует три CSV: `runs_log.csv`, `summary_distance.csv`, `summary_runtime_ms.csv`.

3.6 Критерии остановки и воспроизводимость

- Остановка по *адаптивному* критерию (малое улучшение лучшего).

4 Результаты работы

Результатом программы являются два графика - график получившегося в результате работы алгоритма решения задачи коммивояжера в сравнении с оптимальным решением и график сходимости. **Параметры запуска алгоритма:**

- **Размер популяци:** 220 особей
- **Число поколений:** 200
- **Вероятность кроссинговера:** 0.7
- **Вероятность мутации:** 0.05

Результаты:

- Время работы: 2.08
- Лучшая найденная длина: 27620.8
- Эталонная длина: н/д, будем ориентироваться, на сравнение графиков.

Длина маршрута, сильно схожа, с оптимальным решением. На рисунке 1-5, показаны туры, найденные алгоритмом на покалениях: $\{0, 10, 20, 50, 80\}$.

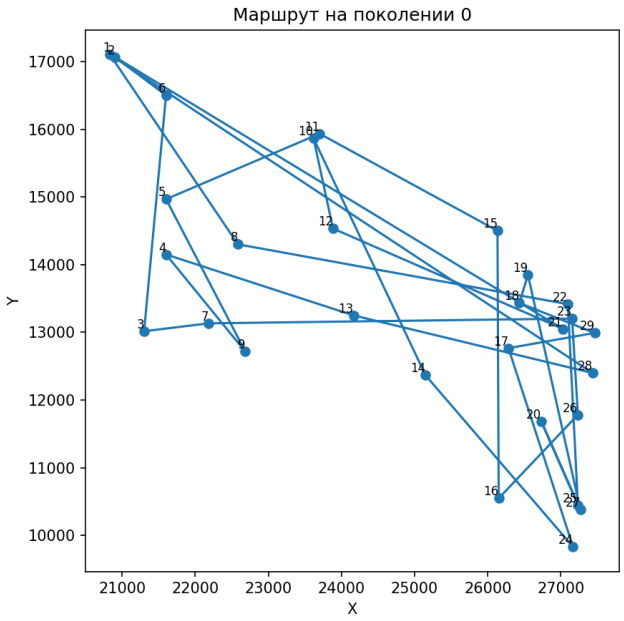

Рис. 1: Тур поколения 0

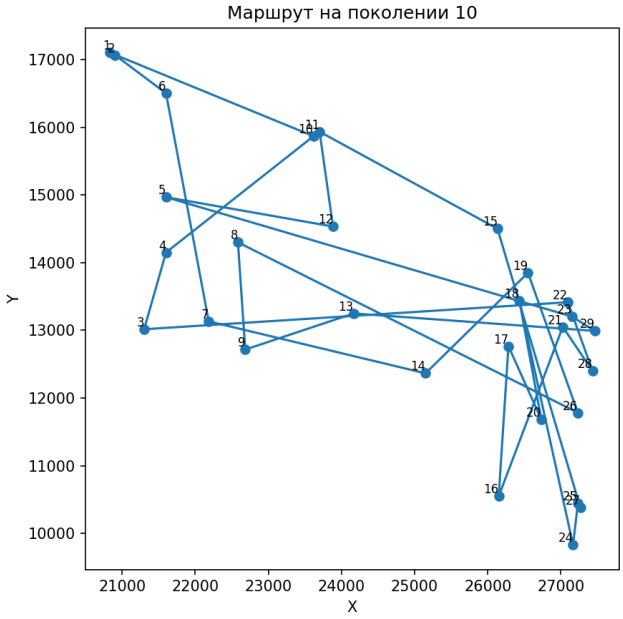

Рис. 2: Тур поколения 10

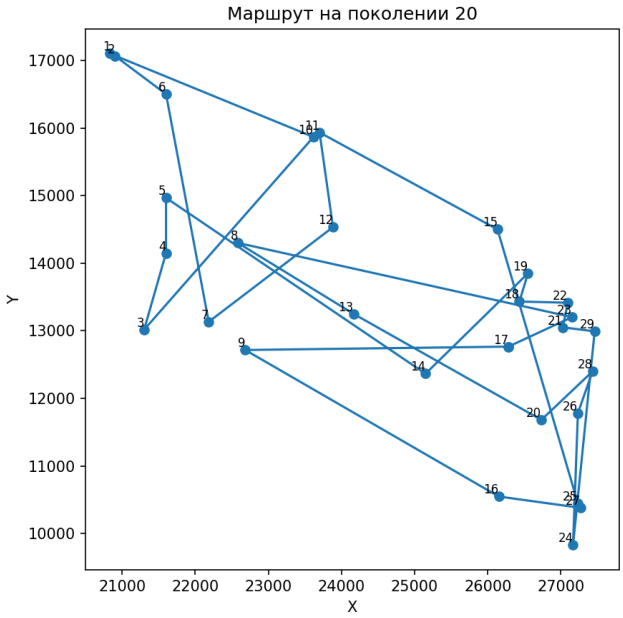

Рис. 3: Тур поколения 20

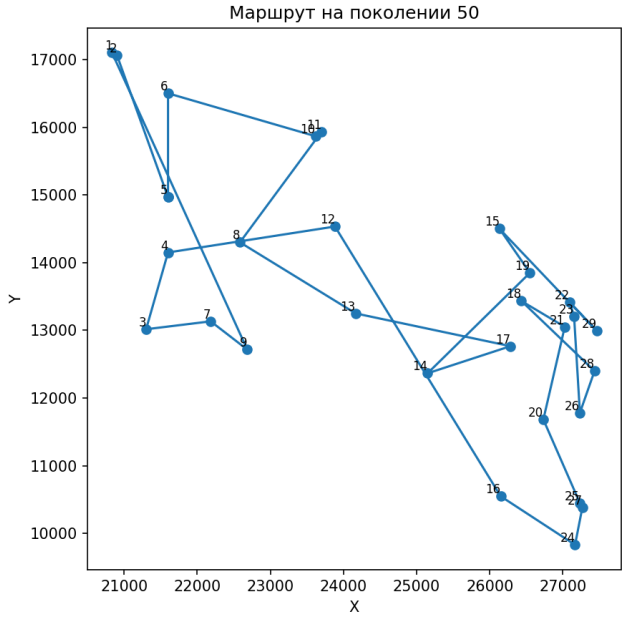

Рис. 4: Тур поколения 50

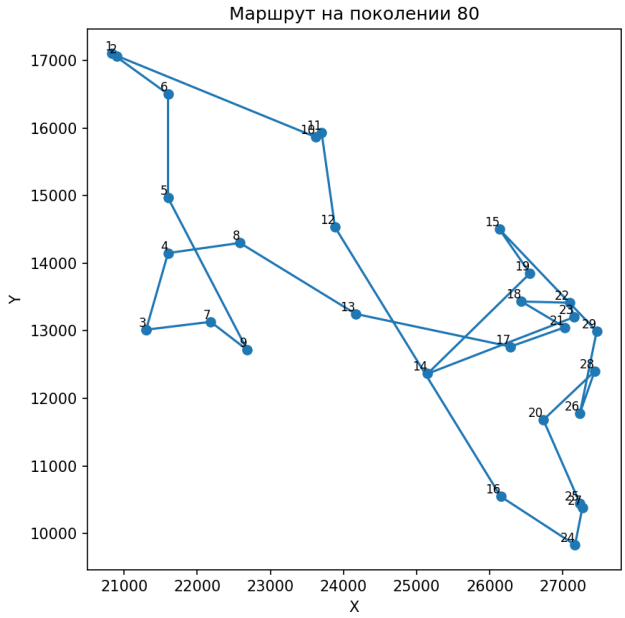

Рис. 5: Тур поколения 80

На рисунке 6 представлен наилучший результат, который перестал улучшатся с 198 поколения.

На рисунке 7, представлен эталлоный оптимальный путь.

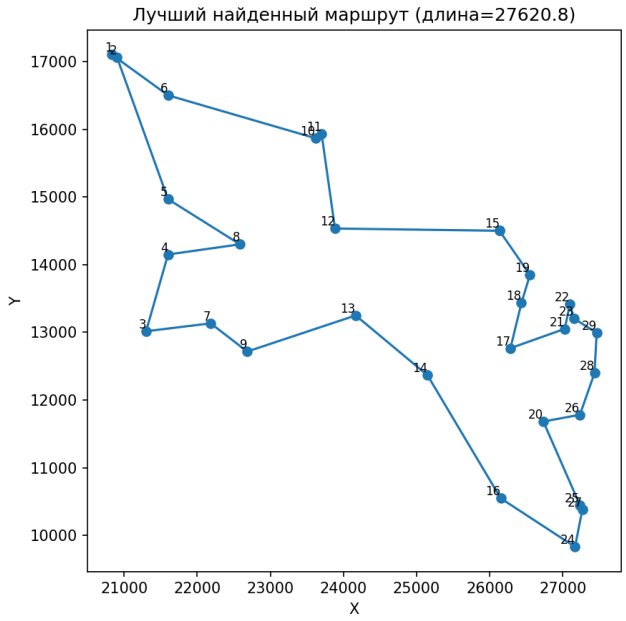

Рис. 6: Лучший результат, поколение 198

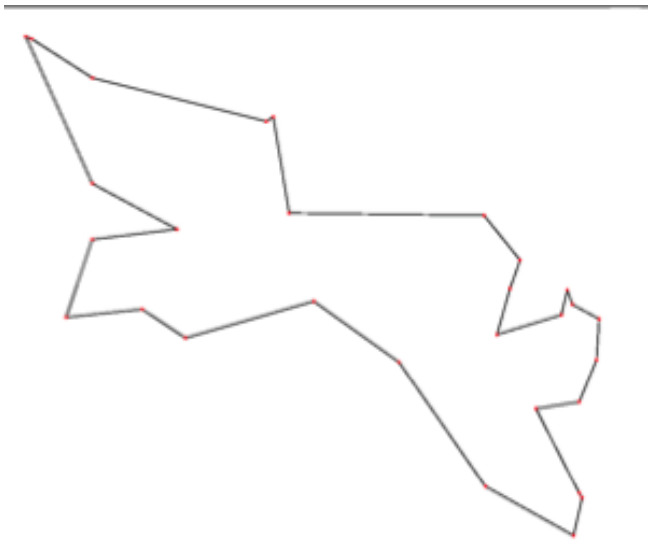

Рис. 7: Эталлоный оптимальный тур

На рисунке 8, представлен график сходимости алгоритма.

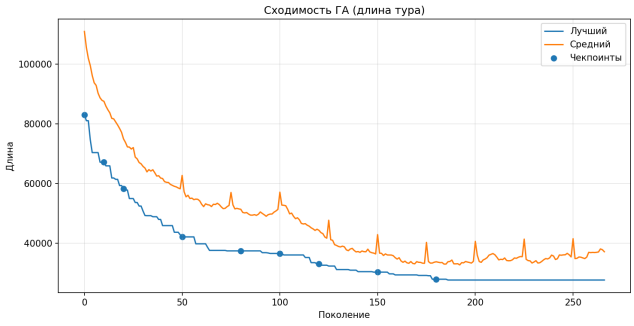

Рис. 8: Сходимость алгоритма

График сходимости демонстрирует последовательную оптимизацию длины маршрута на протяжении всех поколений: в начальной фазе наблюдается интенсивное улучшение решения, которое постепенно замедляется по мере приближения к оптимуму, а в конечных поколениях алгоритм выходит на сходимость - достигнуто оптимальное решение.

5 Анализ результатов и вывод

Дано.

- Размер популяции: 220 особей.
- Наборы вероятностей: $P_c \in \{0.5, 0.7, 0.9\}$, $P_m \in \{0.02, 0.05, 0.10\}$.
- Для каждой конфигурации выполнено 30 запусков; усреднённые метрики собраны в CSV (`summary_distance.csv`, `summary_runtime_ms.csv`).

Требуется: исследовать зависимость качества (длины маршрута) и времени выполнения от P_c и P_m.

5.1 Точность и стабильность решения

В табл. 1 приведены средние длины найденного маршрута (в скобках — стандартное отклонение) по 30 запускам. Лучший результат по качеству выделен красным.

Таблица 1: Средняя дистанция (ст. откл.)

	P_c=0.5	P_c=0.7	P_c=0.9
P_m=0.02	27829.12 (265.12)	27953.25 (467.00)	31171.14 (2988.82)
P_m=0.05	27888.81 (341.77)	27850.12 (226.55)	32685.30 (2594.70)
P_m=0.10	27943.88 (326.52)	27840.98 (333.08)	33936.69 (3333.17)

5.2 Время выполнения

В табл. 2 показано среднее время в секундах (перевод из миллисекунд). Минимальное время выделено красным.

Таблица 2: Среднее время выполнения (с)

	P_c=0.5	P_c=0.7	P_c=0.9
P_m=0.02	1.66	2.27	3.50
P_m=0.05	1.63	2.38	3.67
P_m=0.10	1.62	2.58	4.29

5.3 Анализ результатов

На основе табл. 1–2 при N=220 можно сделать следующие выводы:

- **Лучшее качество** достигается при P_c=0.5, P_m=0.02: 27829.12 ± 265.12 — минимальная средняя длина маршрута среди всех конфигураций.
- Для P_c=0.7 наилучшее качество даёт P_m=0.10 (27840.98), что близко ко всему лучшему, но медленнее по времени.
- При P_c=0.9 наблюдается существенная деградация качества и рост разброса — переизбыточное скрещивание ухудшает решение.
- **Время выполнения** растёт с P_c и слегка уменьшается с ростом P_m; минимум времени у P_c=0.5, P_m=0.10 (1.62 с).
- **Компромисс** «качество–время»: конфигурация P_c=0.5, P_m=0.05 даёт 27888.81 при 1.63 с (почти минимальное время и лишь на ≈ 60 единиц хуже глобального минимума качества).

Одиночный прогон. Контрольный запуск с P_c=0.7, P_m=0.05 дал лучший тур 27620.78 за 266 поколений и 2.082 с, что подтверждает способность алгоритма находить решения лучше средних по сетке.

6 Ответ на контрольный вопрос

Вопрос: Тур в представлении соседства, кроссинговеры обмен ребер, обмен подтуров, эвристический.

Ответ:

Тур в представлении соседства (adjacency). Тур кодируется перестановкой-последователем $\sigma \in S_n$, где $\sigma(i)$ — город, следующий за i.

$$L(\sigma) = \sum_{i=1}^{n} d\big(i, \sigma(i)\big), \qquad \sigma(i) \neq i, \quad |\mathrm{orb}_\sigma(i_0)| = n$$

Последнее условие означает единственный гамильтонов цикл (нет подсубтуров). Эквивалентно матричной модели с индикаторами $x_{ij} \in \{0, 1\}$:

$$\sum_j x_{ij} = 1, \quad \sum_i x_{ij} = 1, \quad L = \sum_{i,j} d_{ij}\, x_{ij}, \quad x_{i,\sigma(i)} = 1.$$

Кроссинговер «обмен рёбер» (Edge Recombination, ERX). Сохраняет рёбра родителей, строя для каждого города список смежности из обоих родителей:

$$\mathcal{N}(i) = \{j : \ (i, j) \in E(P_1) \cup E(P_2)\}.$$

Алгоритм: начинаем с случайного i_1; далее выбираем

$$i_{t+1} = \arg \min_{j \in \mathcal{N}(i_t) \setminus S_t} |\mathcal{N}(j)|,$$

при пустой $\mathcal{N}$ — берём случайный неиспользованный город. Сложность $O(n)$. Сохраняет много «правильных» рёбер.

Кроссинговер «обмен подтуров». В представлении σ выбираем подцикл $C \subseteq V$ у P_1 и копируем его преемников:

$$\sigma_{\text{child}}(i) = \sigma_{P_1}(i) \ \ \forall i \in C.$$

Для $i \notin C$ ставим $\sigma_{\text{child}}(i) = \sigma_{P_2}(i)$, затем устраняем конфликты (двойные входы/выходы) переназначением на ближайших допустимых:

$$\sigma_{\text{child}}(i) = \arg\min_{j \notin S} \ d(i, j) \quad \text{s.t. результирующий граф — единый цикл.}$$

Идея — перенос «хороших» подсубтуров целиком и аккуратное их сшивание.

Эвристический кроссинговер (heuristic/greedy). Формирует ребёнка жадно, используя объединённый набор кандидатных рёбер родителей:

$$\mathcal{C}(i) = \{j : \ (i, j) \in E(P_1) \cup E(P_2)\} \cup \mathrm{KNN}(i).$$

Из текущего города i_t выбираем

$$i_{t+1} = \begin{cases} \arg \min\limits_{j \in \mathcal{C}(i_t) \setminus S_t} d(i_t, j), & \text{если } \mathcal{C}(i_t) \setminus S_t \neq \varnothing, \\ \arg \min\limits_{j \notin S_t} d(i_t, j), & \text{иначе.} \end{cases}$$

Такой оператор одновременно «наследует» структуру родителей и минимизирует локальную длину, часто давая короткие туры и быструю сходимость.

Заключение

В результате выполнения лабораторной работы №2 были достигнуты следующие результаты:

- освоен теоретический материал;
- создана программа на языке `Python` с использованием среды `Jupyter Notebook`;
- проведено исследование влияния параметров генетического алгоритма на эффективность поиска лучшего тура.

Список литературы

1. Методические указания по выполнению лабораторных работ к курсу «Генетические алгоритмы», стр. 119.

```
import time
from dataclasses import dataclass
from typing import Dict, List, Tuple

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

RAW_COORDS = [
    (1, 20833.3333, 17100.0000),
    (2, 20900.0000, 17066.6667),
    (3, 21300.0000, 13016.6667),
    (4, 21600.0000, 14150.0000),
    (5, 21600.0000, 14966.6667),
    (6, 21600.0000, 16500.0000),
    (7, 22183.3333, 13133.3333),
    (8, 22583.3333, 14300.0000),
    (9, 22683.3333, 12716.6667),
    (10, 23616.6667, 15866.6667),
    (11, 23700.0000, 15933.3333),
    (12, 23883.3333, 14533.3333),
    (13, 24166.6667, 13250.0000),
    (14, 25149.1667, 12365.8333),
    (15, 26133.3333, 14500.0000),
    (16, 26150.0000, 10550.0000),
    (17, 26283.3333, 12766.6667),
    (18, 26433.3333, 13433.3333),
    (19, 26550.0000, 13850.0000),
    (20, 26733.3333, 11683.3333),
    (21, 27026.1111, 13051.9444),
    (22, 27096.1111, 13415.8333),
    (23, 27153.6111, 13203.3333),
    (24, 27166.6667, 9833.3333),
    (25, 27233.3333, 10450.0000),
    (26, 27233.3333, 11783.3333),
    (27, 27266.6667, 10383.3333),
    (28, 27433.3333, 12400.0000),
    (29, 27462.5000, 12992.2222),
]
CITY_IDS = [cid for cid, _, _ in RAW_COORDS]
COORDS = np.array([(x, y) for _, x, y in RAW_COORDS], dtype=float)
N = len(COORDS)


def build_distance_matrix(coords: np.ndarray) -> np.ndarray:
    n = coords.shape[0]
    D = np.zeros((n, n), dtype=float)
    for i in range(n):
        diff = coords[i] - coords
        D[i] = np.hypot(diff[:, 0], diff[:, 1])
    return D

DIST = build_distance_matrix(COORDS)

def plot_route(coords: np.ndarray, route: List[int], title: str, filename:
    str | None = None) -> None:
    cyc = route + [route[0]]
    xs = coords[[i for i in cyc], 0]
```

```
    ys = coords[[i for i in cyc], 1]
    plt.figure(figsize=(6, 6))
    plt.plot(xs, ys, marker='o')
    for idx, (x, y) in enumerate(coords):
        plt.text(x, y, str(CITY_IDS[idx]), fontsize=8, ha='right', va='
            bottom')
    plt.title(title)
    plt.xlabel('X')
    plt.ylabel('Y')
    plt.tight_layout()
    if filename:
        plt.savefig(filename, dpi=150)
    plt.close()


def plot_convergence(best: List[float], mean_vals: List[float],
                     checkpoints: Tuple[int, ...], filename: str | None =
                         None) -> None:
    gens = list(range(len(best)))
    plt.figure(figsize=(10, 5.2))
    plt.plot(gens, best, label='best')
    plt.plot(gens, mean_vals, label='mean')
    cps = [cp for cp in checkpoints if cp < len(best)]
    if cps:
        plt.scatter(cps, [best[cp] for cp in cps], marker='o', label='
            check')
    plt.title('dl ture')
    plt.xlabel('pok')
    plt.ylabel('l')
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.tight_layout()
    if filename:
        plt.savefig(filename, dpi=150)
    plt.close()

@dataclass
class GAConfig:
    pop_size: int = 200
    max_gens: int = 2000
    p_cx: float = 0.7
    p_mut: float = 0.05
    elitism: int = 1
    tour_k: int = 2
    keep_history: bool = True
    seed: int | None = None

    immigrants_period: int = 25
    immigrants_ratio: float = 0.10
    sharing_alpha: float = 0.12
    sharing_sample: int = 40

    stop_window: int = 60
    stop_rel_improve: float = 0.003
    stop_gap_rel: float = 0.003
    stop_diversity: float = 0.06
    stop_suspend_after_imm: int = 5
    hard_patience: int = 80


def init_population(n: int, pop_size: int, rng: np.random.Generator) -> np
```

```
    .ndarray:
    pop = np.empty((pop_size, n), dtype=np.int64)
    base = np.arange(n, dtype=np.int64)
    for i in range(pop_size):
        pop[i] = rng.permutation(base)
    return pop


def lengths(pop: np.ndarray, D: np.ndarray) -> np.ndarray:
    nxt = np.roll(pop, -1, axis=1)
    return D[pop, nxt].sum(axis=1)


def tournament_selection(pop: np.ndarray, fit_for_selection: np.ndarray, k
    : int,
                         rng: np.random.Generator) -> np.ndarray:
    m = pop.shape[0]
    idx = rng.integers(0, m, size=(m, k))
    cand = pop[idx]
    cand_fit = fit_for_selection[idx]
    winners = cand[np.arange(m), np.argmin(cand_fit, axis=1)]
    return winners


def ox_crossover(p1: np.ndarray, p2: np.ndarray, rng: np.random.Generator)
    -> np.ndarray:
    n = p1.size
    a, b = sorted(rng.integers(0, n, size=2))
    if a == b:
        b = (a + 1) % n
        if b < a:
            a, b = b, a
    child = -np.ones(n, dtype=np.int64)
    child[a:b+1] = p1[a:b+1]
    used = set(child[a:b+1])
    j = (b + 1) % n
    i = (b + 1) % n
    while (child == -1).any():
        g = p2[j]
        if g not in used:
            child[i] = g
            used.add(g)
            i = (i + 1) % n
        j = (j + 1) % n
    return child


def mutate_inversion(ind: np.ndarray, rng: np.random.Generator) -> None:
    n = ind.size
    i, j = rng.integers(0, n, size=2)
    if i == j:
        j = (j + 1) % n
    if i > j:
        i, j = j, i
    ind[i:j+1] = ind[i:j+1][::-1]
    if rng.random() < 0.1:
        a, b = rng.integers(0, n, size=2)
        if a != b:
            ind[a], ind[b] = ind[b], ind[a]
```

```
def sharing_penalty(pop: np.ndarray, base_lengths: np.ndarray, rng: np.random.Generator,
                    alpha: float, sample_size: int) -> np.ndarray:
    m = pop.shape[0]
    if m <= 1 or alpha <= 0:
        return base_lengths.copy()
    sample_size = min(sample_size, m)
    sample_idx = rng.choice(m, size=sample_size, replace=False)
    sample = pop[sample_idx]

    sim_scores = np.zeros(m, dtype=float)
    for s in sample:
        sim_scores += np.mean(pop == s, axis=1)
    sim_scores /= sample_size
    return base_lengths * (1.0 + alpha * sim_scores)


def mean_hamming_diversity_to_ref(pop: np.ndarray, ref: np.ndarray) -> float:
    sim = np.mean(pop == ref, axis=1)
    return float(1.0 - sim.mean())


def evolve_tsp(D: np.ndarray, cfg: GAConfig) -> Dict[str, object]:
    rng = np.random.default_rng(cfg.seed)
    n = D.shape[0]

    pop = init_population(n, cfg.pop_size, rng)
    L = lengths(pop, D)

    checkpoints = (0, 10, 20, 50, 80, 100, 120, 150, 180)
    best_idx = int(np.argmin(L))
    best_route = pop[best_idx].copy()
    best_len = float(L[best_idx])
    snapshots: Dict[int, List[int]] = {0: best_route.tolist()}

    best_hist: List[float] = [best_len]
    mean_hist: List[float] = [float(L.mean())]

    ref_len = best_len
    no_improve = 0
    last_gen = 0

    gens_since_imm = 1_000_000

    t0 = time.perf_counter()

    for gen in range(1, cfg.max_gens + 1):
        last_gen = gen

        elite_k = max(0, int(cfg.elitism))
        elites = None
        if elite_k > 0:
            elite_idx = np.argpartition(L, elite_k)[:elite_k]
            elites = pop[elite_idx].copy()

        L_for_selection = sharing_penalty(pop, L, rng, cfg.sharing_alpha, cfg.sharing_sample)

        parents = tournament_selection(pop, L_for_selection, cfg.tour_k, rng)
```

```
231
232             rng.shuffle(parents)
233             off = np.empty_like(parents)
234             for i in range(0, cfg.pop_size, 2):
235                 a = parents[i]
236                 b = parents[i + 1 if i + 1 < cfg.pop_size else i]
237                 if rng.random() < cfg.p_cx:
238                     c1 = ox_crossover(a, b, rng)
239                 else:
240                     c1 = a.copy()
241                 if i + 1 < cfg.pop_size:
242                     c2 = ox_crossover(b, a, rng) if rng.random() < cfg.p_cx
                        else b.copy()
243                     off[i], off[i + 1] = c1, c2
244                 else:
245                     off[i] = c1
246
247             mut_mask = rng.random(cfg.pop_size) < cfg.p_mut
248             for r in np.nonzero(mut_mask)[0]:
249                 mutate_inversion(off[r], rng)
250
251             pop = off
252             L = lengths(pop, D)
253
254             if elite_k > 0:
255                 worst_idx = np.argpartition(L, -elite_k)[-elite_k:]
256                 pop[worst_idx] = elites
257                 L = lengths(pop, D)
258
259             did_imm = False
260             if cfg.immigrants_period > 0 and (gen % cfg.immigrants_period ==
                0):
261                 q = int(round(cfg.pop_size * cfg.immigrants_ratio))
262                 if q > 0:
263                     worst_idx = np.argpartition(L, -q)[-q:]
264                     pop[worst_idx] = init_population(n, q, rng)
265                     L = lengths(pop, D)
266                     did_imm = True
267
268             gens_since_imm = 0 if did_imm else gens_since_imm + 1
269
270             cur_best_idx = int(np.argmin(L))
271             cur_best_len = float(L[cur_best_idx])
272             cur_best_route = pop[cur_best_idx].copy()
273
274             best_hist.append(cur_best_len)
275             mean_val = float(L.mean())
276             mean_hist.append(mean_val)
277
278             if gen in checkpoints:
279                 snapshots[gen] = cur_best_route.tolist()
280
281             if cur_best_len < best_len - 1e-12:
282                 best_len = cur_best_len
283                 best_route = cur_best_route
284
285             stop_tripped = False
286             if gens_since_imm >= cfg.stop_suspend_after_imm and gen >= cfg.
                stop_window:
287                 prev_best_win = best_hist[-(cfg.stop_window + 1)]
288                 rel_improve = (prev_best_win - cur_best_len) / max(
```

```
                prev_best_win, 1e-9)
            gap_rel = (mean_val - cur_best_len) / max(cur_best_len, 1e-9)
            diversity = mean_hamming_diversity_to_ref(pop, cur_best_route)

            if (rel_improve < cfg.stop_rel_improve and
                gap_rel < cfg.stop_gap_rel and
                diversity < cfg.stop_diversity):
                stop_tripped = True

        if cur_best_len < ref_len * (1.0 - 1e-6):
            ref_len = cur_best_len
            no_improve = 0
        else:
            no_improve += 1
        if no_improve >= cfg.hard_patience:
            stop_tripped = True

        if stop_tripped:
            last_gen = gen
            break

    t1 = time.perf_counter()

    return {
        "best_route": best_route.tolist(),
        "best_length": best_len,
        "history_best": np.asarray(best_hist),
        "history_mean": np.asarray(mean_hist),
        "snapshots": snapshots,
        "last_generation": last_gen,
        "runtime_sec": (t1 - t0),
    }

def run_grid_and_save_csv(
    D: np.ndarray,
    base_cfg: GAConfig,
    p_c_labels: List[str],
    p_m_labels: List[str],
    runs_per_conf: int = 30,
    seed_base: int = 20251019,
    log_csv_path: str = "runs_log.csv",
    summary_distance_csv_path: str = "summary_distance.csv",
    summary_time_csv_path: str = "summary_runtime_ms.csv",
):
    p_c_map = {lab: float(lab) for lab in p_c_labels}
    p_m_map = {lab: float(lab) for lab in p_m_labels}

    logs = []
    agg_len: Dict[Tuple[str, str], List[float]] = {(pc, pm): [] for pm in
        p_m_labels for pc in p_c_labels}
    agg_time: Dict[Tuple[str, str], List[float]] = {(pc, pm): [] for pm in
        p_m_labels for pc in p_c_labels}

    for pm_lab, pm_val in p_m_map.items():
        for pc_lab, pc_val in p_c_map.items():
            for r in range(runs_per_conf):
                seed = seed_base ^ (hash((pc_lab, pm_lab, r)) & 0x7FFFFFFF
                    )
                cfg = GAConfig(
                    pop_size=base_cfg.pop_size,
                    max_gens=base_cfg.max_gens,
```

```
                    p_cx=pc_val,
                    p_mut=pm_val,
                    elitism=base_cfg.elitism,
                    tour_k=base_cfg.tour_k,
                    keep_history=False,
                    seed=seed,
                    immigrants_period=base_cfg.immigrants_period,
                    immigrants_ratio=base_cfg.immigrants_ratio,
                    sharing_alpha=base_cfg.sharing_alpha,
                    sharing_sample=base_cfg.sharing_sample,
                    stop_window=base_cfg.stop_window,
                    stop_rel_improve=base_cfg.stop_rel_improve,
                    stop_gap_rel=base_cfg.stop_gap_rel,
                    stop_diversity=base_cfg.stop_diversity,
                    stop_suspend_after_imm=base_cfg.stop_suspend_after_imm,
                    hard_patience=base_cfg.hard_patience,
                )
                res = evolve_tsp(D, cfg)
                logs.append({
                    "seed": seed,
                    "P_c_label": pc_lab,
                    "P_m_label": pm_lab,
                    "P_c": pc_val,
                    "P_m": pm_val,
                    "best_length": res["best_length"],
                    "generations": res["last_generation"],
                    "runtime_ms": res["runtime_sec"] * 1000.0
                })
                agg_len[(pc_lab, pm_lab)].append(res["best_length"])
                agg_time[(pc_lab, pm_lab)].append(res["runtime_sec"] *
                    1000.0)

    pd.DataFrame(logs).to_csv(log_csv_path, index=False, encoding="utf-8")

    rows = []
    for pm_lab in p_m_labels:
        row = []
        for pc_lab in p_c_labels:
            vals = np.asarray(agg_len[(pc_lab, pm_lab)], dtype=float)
            mean = float(vals.mean()) if vals.size else float("nan")
            std = float(vals.std(ddof=1)) if vals.size > 1 else 0.0
            row.append(f"{mean:.2f} + {std:.2f}")
        rows.append(row)
    df_dist = pd.DataFrame(rows,
                           index=[f"P_m={pm}" for pm in p_m_labels],
                           columns=[f"P_c={pc}" for pc in p_c_labels])
    df_dist.to_csv(summary_distance_csv_path, encoding="utf-8")

    rows_t = []
    for pm_lab in p_m_labels:
        row_t = []
        for pc_lab in p_c_labels:
            vals_t = np.asarray(agg_time[(pc_lab, pm_lab)], dtype=float)
            row_t.append(float(vals_t.mean()) if vals_t.size else float("
                nan"))
        rows_t.append(row_t)
    df_time = pd.DataFrame(rows_t,
                           index=[f"P_m={pm}" for pm in p_m_labels],
                           columns=[f"P_c={pc}" for pc in p_c_labels])
    df_time.to_csv(summary_time_csv_path, encoding="utf-8")
```

```
404
405         print(f"- {log_csv_path}")
406         print(f"- {summary_distance_csv_path}")
407         print(f"- {summary_time_csv_path}")
408
409     def main():
410         base = GAConfig(
411             pop_size=220,
412             max_gens=2000,
413             p_cx=0.7,
414             p_mut=0.05,
415             elitism=1,
416             tour_k=2,
417             keep_history=True,
418             seed=42,
419
420             immigrants_period=25,
421             immigrants_ratio=0.10,
422             sharing_alpha=0.12,
423             sharing_sample=40,
424
425             stop_window=60,
426             stop_rel_improve=0.003,
427             stop_gap_rel=0.003,
428             stop_diversity=0.06,
429             stop_suspend_after_imm=5,
430             hard_patience=80,
431         )
432
433         res = evolve_tsp(DIST, base)
434         print(f": {res['best_length']:.3f}")
435         print(f": {res['last_generation']}")
436         print(f": {res['runtime_sec']:.3f} sec")
437
438         best_hist = res["history_best"]
439         mean_hist = res["history_mean"]
440         plot_convergence(best_hist.tolist(), mean_hist.tolist(),
441                          (0, 10, 20, 50, 80, 100, 120, 150, 180),
442                          filename="convergence.png")
443
444         for cp, route in res["snapshots"].items():
445             plot_route(COORDS, route, f"Rout {cp}", filename=f"route_gen_{cp}.
                png")
446
447         plot_route(COORDS, res["best_route"],
448                    f"(l={res['best_length']:.1f})",
449                    filename="best_route.png")
450
451         pd.DataFrame([{
452             "best_length": res["best_length"],
453             "generations": res["last_generation"],
454             "runtime_ms": res["runtime_sec"] * 1000.0,
455             "pop_size": base.pop_size,
456             "P_c": base.p_cx,
457             "P_m": base.p_mut
458         }]).to_csv("single_run_summary.csv", index=False, encoding="utf-8")
459         print("[OK] single_run_summary.csv")
460
461         p_c_labels = ["0.5", "0.7", "0.9"]
462         p_m_labels = ["0.02", "0.05", "0.10"]
463         run_grid_and_save_csv(
```

```
            DIST, base,
            p_c_labels=p_c_labels,
            p_m_labels=p_m_labels,
            runs_per_conf=30,
            seed_base=123456,
            log_csv_path="runs_log.csv",
            summary_distance_csv_path="summary_distance.csv",
            summary_time_csv_path="summary_runtime_ms.csv"
        )

if __name__ == "__main__":
    main()
```