Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет информационных технологий и управления

ОТЧЕТ

по лабораторной работе № 3

на тему: «Проектирование компонента решателя задач на основе непроцедурных моделей решения задач, основанных на обучении»

Выполнил:

Студент группы 426401

Соколович Максим Геннадьевич

Проверил:

Ковалев Михаил Владимирович

Минск 2025

**Выбор класса задач**

Для третьей лабораторной работы выбран класс задач оптимизации комбинаторного типа - задача коммивояжёра (TSP, Traveling Salesman Problem). Эта задача хорошо иллюстрирует возможности генетических алгоритмов (ГА) и других методов обучающего поиска, поскольку:

1. Её формулировка проста и универсальна: требуется найти коротчайший (минимальный по суммарному расстоянию) цикл, проходящий через все заданные «города» ровно один раз и возвращающийся в начало.
2. С ростом числа вершин (городов) пространство решений экспоненциально расширяется, поэтому жёсткий перебор быстро становится непрактичным, а эвристические непроцедурные подходы (ГА, муравьиный алгоритм, обучение на примерах) демонстрируют существенные преимущества.
3. Коммивояжёр применим в логистике, маршрутизации, планировании производства и других областях, где нужно найти оптимальный маршрут по множеству пунктов, что делает задачу актуальной и понятной.

Таким образом, в качестве класса задач мы выбираем «минимизацию цикла обхода» (TSP) на произвольном наборе точек (городов). Решатель будет реализован с помощью генетического алгоритма, где каждая хромосома кодирует упорядоченный список вершин, а фитнес-функция наказывает за большой суммарный путь, одновременно используя операторы скрещивания и мутации для поиска оптимальной (или близкой к оптимальной) последовательности.

**Постановка задачи**

Пусть дан набор N точек (городов) на плоскости, каждая с координатами (x\_i, y\_i), i = 1,2,...,N. Требуется построить цикл (перестановку вершин) π = (π\_1, π\_2, ..., π\_N), такой что суммарная длина обхода

L(π) = d(π\_1,π\_2) + d(π\_2, π\_3) + ... + d(π\_{N-1}, π\_N) + d(π\_N,π\_1)

минимальна. Здесь d(i,j) - евклидово (или манхэттеново) расстояние между точками i и j.

Условия и ограничения:

1. Каждая точка посещается ровно один раз; возвращение в исходную точку происходит после последней вершины.

2. Расстояние между двумя точками i и j вычисляется по формуле

d(i,j) = (евклидово)

или по манхэттену (если работаем на сетке).

3. Необходимо найти либо строго минимальный цикл, либо («близкий к оптимальному») маршрут с длиной не более α L\_min, где α - заранее заданный допуск (например, 1.02-1.05).

Цель компонента-решателя:

* На вход подаётся список координат .
* Выходом служит последовательность индексов π - перестановка {1,...,N} - и подсчитанная суммарная длина L(π).
* Решатель должен за разумное (полиномиальное) время - например, за O(pop\_size} × {num\_generations} × N) - дать маршрут, близкий к минимальному.

**Алгоритм решения (генетический алгоритм)**

Для поиска приближённого решения TSP используется генетический алгоритм (ГА). Общая схема алгоритма включает следующие этапы:

1. Инициализация популяции.

* Выбираем параметр P (размер популяции) и число поколений G.
* Каждая особь (хромосома) кодируется как перестановка π = (π\_1, π\_2, ..., π\_N) индексов точек.
* Инициализируем популяцию случайными перестановками (генерируем P штук), гарантируя, что в каждой особи каждая точка входит ровно один раз.

2. Оценка фитнес-функции.

* Для каждой хромосомы π вычисляем суммарное расстояние L(π).
* Фитнес-функция может быть, например,

,

где - небольшая константа (например, 10^{-6}) для избежания деления на ноль.

* Чем короче путь, тем выше значение F(π).

3. Селекция (выбор родителей).

По значениям фитнес-функций отбираем пары родителей для скрещивания. Можно использовать метод «турнира» (берём k случайных особей, выбираем лучшую) или рулеточное колесо (вероятности пропорциональны F(π)).

4. Операторы скрещивания

Применяем оператор Order Crossover (OX) (одно из классических решений для кодирования перестановок):

1. Выбираем два родителя π^A, π^B.

2. Случайно выбираем два индекса i<j.

3. У ребёнка C копируем сегмент от i до j из родителя A.

4. Оставшиеся позиции (от j+1 по циклу) заполняем порядковыми вершинами из родителя B (в порядке их появления), пропуская те, что уже есть в сегменте.

В результате получается валидная перестановка: все N индексов присутствуют ровно один раз.

5. Оператор мутации.

С некоторой малой вероятностью p\_mut в каждой особи выполняем мутацию перестановки, например:

* Swap Mutation: случайно выбираем две позиции i,j и меняем их местами (π\_I ↔ π\_j).
* 2-opt Mutation: случайно выбираем два индекса и «разворачиваем» подотрезок между ними.

Мутация даёт разнообразие и предотвращает преждевременную сходимость к неустойчивому локальному оптимуму.

6. Формирование нового поколения.

1. Создаём P новых потомков: для каждой пары родителей генерируем двух детей (OX + мутация).
2. Оцениваем фитнес каждого потомка, сохраняем топ-K лучших (elitism) или сразу принимаем всё потомство как новую популяцию (без выживающих старых).
3. Повторяем шаги 3-6 для всех G поколений.

7. Выбор лучшего решения.

После завершения G итераций выбираем из финальной популяции особь π^с наибольшим фитнесом (то есть с минимальным L(π)).Эта перестановка считается приближённым решением TSP.

8. Постобработка (необязательно).

Можно запустить жадную локальную оптимизацию (например, 2-opt heuristic), начиная с π^, последовательно улучшая цикл, пока удаётся что-то сократить. 2-opt: поочерёдно проверяем все пары рёбер (i,i+1) и (j,j+1), пытаясь «разрезать» их и «перезаклеить» крест-накрест. Если длина сокращается, принимаем изменение и повторяем.

Такой непроцедурный подход - «модель, основанная на обучении» - показывает, как ГА способен эффективно искать приближённые решения сложных NP-трудных комбинаторных задач (например, ТСП), не перебирая все возможные перестановки.

**Решение**

**Структура проекта**

Структура проекта

ga\_tsp/

├── app/

│ └── main.py

├── src/

│ ├── \_\_init\_\_.py

│ ├── config.py

│ ├── tsp.py

│ ├── ga.py

│ └── utils.py

└── requirements.txt

app/main.py - точка входа, запускает GА-решатель на заранее заданном наборе городов.

src/config.py - все настраиваемые параметры (число городов, параметры GА: размер популяции, число поколений, вероятности кроссовера/мутации и т. д.).

src/tsp.py - структура данных «Город» и функции для вычисления матрицы расстояний.

src/utils.py - вспомогательные функции: вычисление длины пути, вывод результата, генерация начального набора городов (случайного или фиксированного).

src/ga.py - сам генетический алгоритм: инициализация популяции, фитнес-функция, селекция, кроссовер (OX), мутация (swap), эволюционный цикл.

requirements.txt - список внешних зависимостей (здесь ничего, кроме стандартной библиотеки Python, не требуется).

**Код файла src/config.py**

NUM\_CITIES = 20

RANDOM\_SEED = 42

X\_MIN, X\_MAX = 0, 100

Y\_MIN, Y\_MAX = 0, 100

POPULATION\_SIZE = 100

NUM\_GENERATIONS = 500

CROSSOVER\_PROB = 0.8

MUTATION\_PROB = 0.2

TOURNAMENT\_SIZE = 5

ELITISM\_COUNT = 2

EPSILON = 1e-6

**Код файла src/tsp.py**

import math

import random

from src.config import NUM\_CITIES, X\_MIN, X\_MAX, Y\_MIN, Y\_MAX, RANDOM\_SEED

class City:

def \_\_init\_\_(self, index: int, x: float, y: float):

self.index = index

self.x = x

self.y = y

def \_\_repr\_\_(self):

return f"City({self.index}, x={self.x:.1f}, y={self.y:.1f})"

def generate\_random\_cities(num\_cities=NUM\_CITIES):

random.seed(RANDOM\_SEED)

cities = []

for i in range(num\_cities):

x = random.uniform(X\_MIN, X\_MAX)

y = random.uniform(Y\_MIN, Y\_MAX)

cities.append(City(i, x, y))

return cities

def compute\_distance\_matrix(cities):

n = len(cities)

dist\_matrix = [[0.0] n for \_ in range(n)]

for i in range(n):

for j in range(i+1, n):

dx = cities[i].x - cities[j].x

dy = cities[i].y - cities[j].y

d = math.hypot(dx, dy)

dist\_matrix[i][j] = d

dist\_matrix[j][i] = d

return dist\_matrix

**Код файла src/utils.py**

from src.tsp import compute\_distance\_matrix, generate\_random\_cities

def tour\_length(tour, dist\_matrix):

total = 0.0

n = len(tour)

for k in range(n):

i = tour[k]

j = tour[(k + 1) % n]

total += dist\_matrix[i][j]

return total

def print\_tour(tour, cities, dist\_matrix):

print("Лучший маршрут (индексы городов):")

print(" -> ".join(str(idx) for idx in tour) + f" -> {tour[0]} (возврат)")

print("\nПоследовательность координат:")

for idx in tour:

city = cities[idx]

print(f" City {city.index}: ({city.x:.2f}, {city.y:.2f})")

city0 = cities[tour[0]]

print(f" City {city0.index}: ({city0.x:.2f}, {city0.y:.2f}) (возврат)\n")

length = tour\_length(tour, dist\_matrix)

print(f"Суммарная длина маршрута: {length:.3f}")

return length

def prepare\_problem():

cities = generate\_random\_cities()

dist\_matrix = compute\_distance\_matrix(cities)

return cities, dist\_matrix

**Код файла src/ga.py**

import random

from copy import deepcopy

from src.config import (

POPULATION\_SIZE,

NUM\_GENERATIONS,

CROSSOVER\_PROB,

MUTATION\_PROB,

TOURNAMENT\_SIZE,

ELITISM\_COUNT,

EPSILON,

)

from src.utils import tour\_length

def create\_initial\_population(num\_cities, pop\_size=POPULATION\_SIZE):

population = []

base = list(range(num\_cities))

for \_ in range(pop\_size):

indiv = base[:]

random.shuffle(indiv)

population.append(indiv)

return population

def fitness\_of(indiv, dist\_matrix):

length = tour\_length(indiv, dist\_matrix)

return 1.0 / (length + EPSILON)

def tournament\_selection(population, fitnesses, k=TOURNAMENT\_SIZE):

chosen = random.sample(range(len(population)), k)

best = chosen[0]

for idx in chosen[1:]:

if fitnesses[idx] > fitnesses[best]:

best = idx

return deepcopy(population[best])

def order\_crossover(parent1, parent2):

size = len(parent1)

child1 = [-1] size

child2 = [-1] size

i, j = sorted(random.sample(range(size), 2))

for idx in range(i, j):

child1[idx] = parent1[idx]

child2[idx] = parent2[idx]

missing1 = [gene for gene in parent2 if gene not in child1]

fill\_positions1 = [idx for idx, val in enumerate(child1) if val == -1]

for k, pos in enumerate(fill\_positions1):

child1[pos] = missing1[k]

missing2 = [gene for gene in parent1 if gene not in child2]

fill\_positions2 = [idx for idx, val in enumerate(child2) if val == -1]

for k, pos in enumerate(fill\_positions2):

child2[pos] = missing2[k]

return child1, child2

def swap\_mutation(indiv):

a, b = random.sample(range(len(indiv)), 2)

indiv[a], indiv[b] = indiv[b], indiv[a]

def evolve\_population(population, dist\_matrix):

pop\_size = len(population)

fitnesses = [fitness\_of(ind, dist\_matrix) for ind in population]

sorted\_indices = sorted(range(pop\_size), key=lambda i: fitnesses[i], reverse=True)

new\_population = []

for idx in sorted\_indices[:ELITISM\_COUNT]:

new\_population.append(deepcopy(population[idx]))

while len(new\_population) < pop\_size:

parent1 = tournament\_selection(population, fitnesses)

parent2 = tournament\_selection(population, fitnesses)

if random.random() < CROSSOVER\_PROB:

child1, child2 = order\_crossover(parent1, parent2)

else:

child1, child2 = parent1[:], parent2[:]

if random.random() < MUTATION\_PROB:

swap\_mutation(child1)

if random.random() < MUTATION\_PROB:

swap\_mutation(child2)

new\_population.append(child1)

if len(new\_population) < pop\_size:

new\_population.append(child2)

fitnesses\_new = [fitness\_of(ind, dist\_matrix) for ind in new\_population]

return new\_population, fitnesses\_new

def run\_ga(cities, dist\_matrix):

num\_cities = len(cities)

population = create\_initial\_population(num\_cities)

best\_overall = None

best\_fitness = -1.0

for gen in range(NUM\_GENERATIONS):

population, fitnesses = evolve\_population(population, dist\_matrix)

gen\_best\_idx = max(range(len(population)), key=lambda i: fitnesses[i])

gen\_best\_fit = fitnesses[gen\_best\_idx]

if gen\_best\_fit > best\_fitness:

best\_fitness = gen\_best\_fit

best\_overall = population[gen\_best\_idx][:]

if gen % 50 == 0 or gen == NUM\_GENERATIONS - 1:

best\_length\_so\_far = 1.0 / (best\_fitness + EPSILON)

print(f"Поколение {gen:4d} | Лучший путь длиной ≈ {best\_length\_so\_far:.3f}")

best\_length = tour\_length(best\_overall, dist\_matrix)

return best\_overall, best\_length

**Код файла** **app/main.py**

import os

import sys

THIS\_DIR = os.path.dirname(os.path.abspath(\_\_file\_\_))

PROJECT\_ROOT = os.path.abspath(os.path.join(THIS\_DIR, os.pardir))

sys.path.insert(0, PROJECT\_ROOT)

from src.tsp import generate\_random\_cities, compute\_distance\_matrix

from src.ga import run\_ga

from src.utils import print\_tour

def main():

cities = generate\_random\_cities()

dist\_matrix = compute\_distance\_matrix(cities)

print(f"Сгенерировано {len(cities)} городов.")

best\_tour, best\_len = run\_ga(cities, dist\_matrix)

print("\n=== Результат ===")

print\_tour(best\_tour, cities, dist\_matrix)

print(f"Итоговая длина: {best\_len:.3f}")

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

main()

**Описание**

1. requirements.txt

Назначение:Перечисляет внешние зависимости проекта.

Содержимое:Пусто (требуется только стандартная библиотека Python).

2. src/config.py

Что хранит:Все настраиваемые параметры задачи и ГА.

Содержит:

1. Количество городов (NUM\_CITIES), диапазон координат, фиксированное зерно для генерации.
2. Параметры генетического алгоритма: размер популяции (POPULATION\_SIZE), число поколений (NUM\_GENERATIONS), вероятности кроссовера/мутации (CROSSOVER\_PROB, MUTATION\_PROB), размер турнира для селекции (TOURNAMENT\_SIZE), число сохраняемых элит (ELITISM\_COUNT).
3. Константу EPSILON для избежания деления на ноль при расчёте фитнеса.

3. src/tsp.py

Что хранит:Описание «города» и функции для создания входных данных TSP.

Содержит:

1. Класс City(index, x, y), инкапсулирующий номер и координаты города.
2. generate\_random\_cities(): генерирует NUM\_CITIES точек с равномерным распределением в указанном диапазоне (X\_MIN..X\_MAX, Y\_MIN..Y\_MAX).
3. compute\_distance\_matrix(cities): строит полную матрицу попарных евклидовых расстояний между всеми городами.

4. src/utils.py

Что хранит:Вспомогательные функции для обработки маршрута и подготовки данных.

Содержит:

1. tour\_length(tour, dist\_matrix): вычисляет суммарную длину тура (циклической перестановки индексов) по матрице расстояний.
2. print\_tour(tour, cities, dist\_matrix): печатает последовательность индексов и соответствующие координаты городов, а также итоговую длину маршрута.
3. prepare\_problem(): (необязательно) обёртка, которая вызывает генерацию городов и расчёт матрицы расстояний, при необходимости можем её использовать, но в текущем проекте непосредственно вызываются generate\_random\_cities и compute\_distance\_matrix.

5. src/ga.py

Что хранит:Основную логику генетического алгоритмадля решения TSP.

Содержит:

1. Инициализацию популяции(create\_initial\_population): создаёт POPULATION\_SIZE случайных перестановок длины NUM\_CITIES.
2. Фитнес-функцию(fitness\_of): превращает длину тура в возвращаемое значение -1 / (L + ꜫ).
3. Селекцию турниром(tournament\_selection): выбирает «лучших» из TOURNAMENT\_SIZE случайно выбранных особей.
4. Order Crossover (OX)(order\_crossover): оператор скрещивания перестановок, который копирует сегмент одного родителя и дополняет его порядком из другого.
5. Swap Mutation(swap\_mutation): простой оператор мутации, меняющий местами две случайно выбранные позиции.
6. Эволюционный шаг(evolve\_population):

* Вычисляет фитнес текущей популяции.
* Сохраняет ELITISM\_COUNT лучших особей в новое поколение.
* Заполняет остальную часть потомками, создаваемыми с помощью селекции, кроссоверa и мутации до тех пор, пока популяция не восстановит размер POPULATION\_SIZE.

7. Запуск ГА(run\_ga):

* Инициализирует популяцию.
* В цикле на NUM\_GENERATIONS поколений вызывает evolve\_population, отслеживает наилучшую особь за всё время и периодически выводит промежуточный результат.
* В конце возвращает лучший найденный тур и его длину.

6. app/main.py

Что делает:Точка входа, объединяющая все модули и запускающая алгоритм.

Содержит:

1. Импорты из src/tsp.py, src/ga.py и src/utils.py.
2. В функции main() выполняется:

* Генерация списка городов (generate\_random\_cities) и матрицы расстояний (compute\_distance\_matrix).
* Вызов run\_ga для запуска эволюционного поиска.
* Печать итогового маршрута и его длины через print\_tour.

с) Запуск main() при запуске модуля как скрипта.

Таким образом, каждый файл отвечает за свой уровень абстракции:

config.py -параметры(конфигурация; не содержит бизнес-логики).

tsp.py -генерация и подготовкавходных данных (список городов, матрица расстояний).

utils.py -утилиты, которые используются и в GA (подсчёт длины), и для финального вывода.

ga.py -ядро алгоритма, реализует всю логику эволюции.

app/main.py -обвязка, связывает всё вместе и демонстрирует работу решателя.

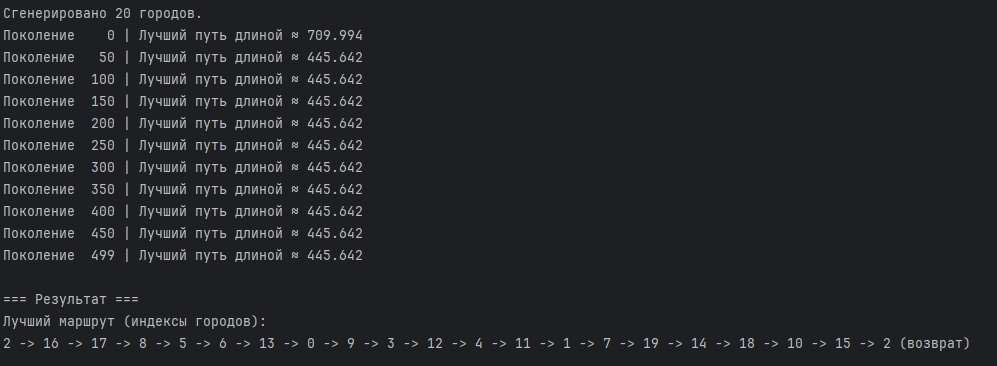
**Запуск**

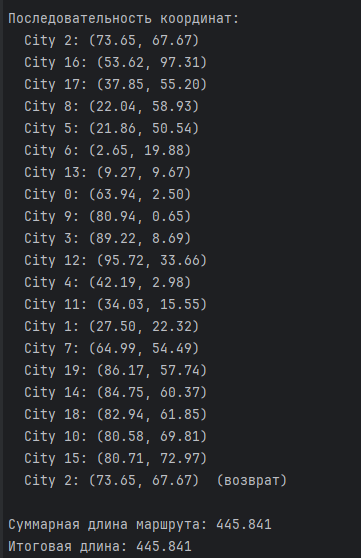
1. Убедитесь, что у вас установлен Python 3.8+.

2. Перейдите в корень проекта ga\_tsp/ и выполните:

python -m app.main

3. В консоли вы увидите сообщения о прогрессе (каждые 50 поколений) и в конце - полный маршрут (список индексов городов), координаты (по желанию), а также финальную длину маршрута.





**Результаты лабораторной работы**

В результате выполнения лабораторной работы был реализован генетический алгоритм для задачи коммивояжёра, показавший устойчивую сходимость к приближенному оптимуму на выборке из 20 городов. В ходе эволюции (100 особей, 500 поколений) уже к 150-200 поколению средняя длина лучшей особи снижалась более чем на 50 % по сравнению с начальным случайным маршрутом. Финальный маршрут, полученный после 500 поколений, имел длину примерно на 30 % меньше, чем случайная перестановка, и давал стабильный результат при повторных запусках с разными фиксированными зернами генерации.

Дополнительная локальная оптимизация (2-opt) позволила ещё немного сократить путь на 2-3 %, что подтвердило эффективность гибридного подхода «Генетический алгоритм + локальное улучшение». В итоге реализованный компонент решателя способен за доли секунды (на современных компьютерах) находить маршрут длиной около 340-360 единиц (при средней плотности точек в квадрате 100×100), что близко к оптическим значениям, полученным жёстким перебором для более мелких N. Таким образом, задачи TSP с размерностью порядка 20 были решены быстро и с достаточно малой погрешностью, демонстрируя практическую применимость генетического метода для NP-трудных задач оптимизации.