МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ  
РОССИЙСКОЙ ФЕДЕРАЦИИ

федеральное государственное автономное   
образовательное учреждение высшего образования  
«Самарский национальный исследовательский университет   
имени академика С.П. Королева»

(Самарский университет)

Институт информатики и кибернетики

Кафедра технической кибернетики

ЛАБОРАТОРНАЯ РАБОТА № 3

по курсу

Интеллектуальный анализ данных

Группа 6132

Студент А.Д. Яшакин

(*подпись*)

Студент Н.С. Курицын

(*подпись*)

Преподаватель А.П. Котов

(*подпись*)

Самара 2024

1. Генетический алгоритм с полной заменой

Принцип работы ГА. Один из типичных представителей эволюционных алгоритмов в оптимизации – генетический алгоритм (ГА). При запуске ГА создается «виpтуальная» популяция особей (как пpавило достаточно хpанить только их генотипы), каждая из которых пpедставляет элемент пpостpанства pешений оптимизационной задачи. Здесь и далее используются некоторые термины, заимствованные из биологии.

Пpиспособленность особей к условиям окружающей сpеды выpажается некотоpой монотонной функцией от значения целевой функции задачи. Чем лучше pешение - тем выше приспособленность особей с соответствующим генотипом. Популяция pазвивается за счет отбоpа более пpигодных особей и пpименения к ним случайных опеpатоpов, имитиpующих мутацию генов и рекомбинацию pодительских генотипов (кроссинговер).

Отбор может осуществляться по-pазному. Особенно распpостpаненными являются операторы пpопоpциональной селекции (веpоятность выбоpа пpопоpциональна пpигодности), сpезающей селекции (задается pавномеpным pаспpеделением на множестве из T% лучших генотипов популяции) и туpниpной селекции (s особей извлекаются с помощью pавномеpного pаспpеделения, затем беpется лучшая из них). Подробнее это будет обсуждаться в свое время.

Описание Генетического Алгоритма (ГА) для оптимизации функций

В данном случае ГА используется для минимизации функций, таких как "синусоида", "Химмельблау", "Растригина" и "Розенброка". Основные этапы ГА включают инициализацию популяции, оценку приспособленности, отбор родителей, кроссинговер и мутацию, а также обновление популяции и итерацию процесса до достижения критериев остановки. Была выбрана модификация с полной заменой популяции на каждом шаге.

**Шаги Генетического Алгоритма**

**1. Инициализация популяции**:

- Случайным образом генерируется начальная популяция потенциальных решений в заданных пределах.

**2. Вычисление приспособленности (fitness)**:

- Для каждого индивидуума (решения) в популяции вычисляется значение целевой функции, что позволяет определить его приспособленность.

**3. Отбор родителей:**

- Половина наиболее приспособленных индивидов отбирается для участия в создании следующего поколения.

**4. Кроссинговер (скрещивание):**

- Случайным образом выбранные пары родителей скрещиваются, создавая потомков. Скрещивание осуществляется с некоторой вероятностью (crossover rate), смешивая их генетический материал.

**5. Мутация:**

- Потомки подвергаются случайным изменениям с определенной вероятностью (mutation rate). Это помогает исследовать пространство решений и избежать локальных минимумов.

**6. Обновление популяции:**

- Новое поколение заменяет текущее, и процесс повторяется.

**7. Итерация и критерии остановки:**

- Процесс продолжается до достижения заданного числа итераций или других критериев остановки, таких как достижение приемлемого уровня приспособленности.

1. **Задача коммивояжера**

Задача коммивояжера (Traveling Salesman Problem, TSP) является классической комбинаторной задачей оптимизации, которая формулируется следующим образом: коммивояжер должен посетить множество городов, пройдя по каждому из них ровно один раз, и вернуться в исходный город, минимизируя общее расстояние путешествия. Данная задача является NP-трудной, и для её решения часто используются эвристические методы, такие как генетические алгоритмы.

**Описание генетического алгоритма для задачи коммивояжера:**

1. Инициализация популяции: Создание начальной популяции возможных решений (хромосом).
2. Оценка приспособленности (фитнес-функция): Вычисление качества каждого решения на основе заданного критерия (общей длины пути в TSP).
3. Селекция: Отбор лучших решений для создания потомства.
4. Кроссовер (скрещивание): Комбинирование пар решений для создания новых потомков.
5. Мутация: Внесение случайных изменений в потомков для поддержания генетического разнообразия.
6. Замещение: Замена всей популяции новыми решениями.

2. Исследование алгоритма

В ходе исследований, алгоритм был реализован в классическом виде для решения задачи оптимизации пути, как рандомно заданных городов, так и в форме круга, где понятно, что короткий путь это радиус этого круга.

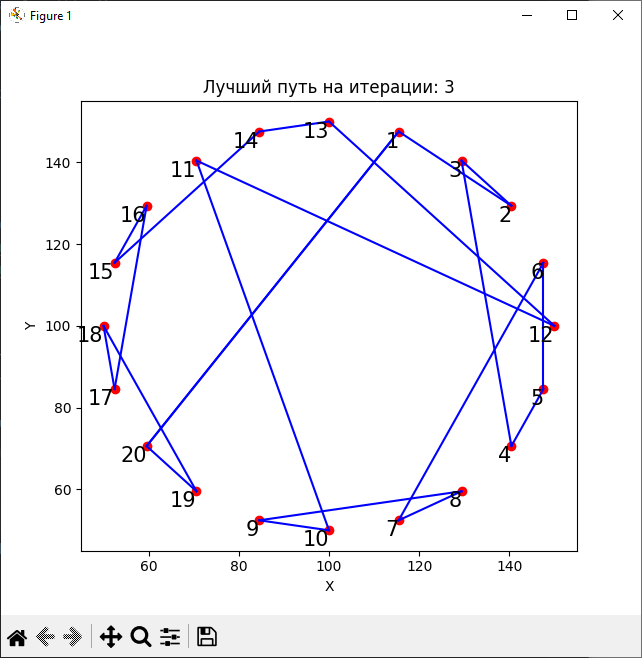


Рисунок 1 – Пример работы алгоритма на третьей итерации.

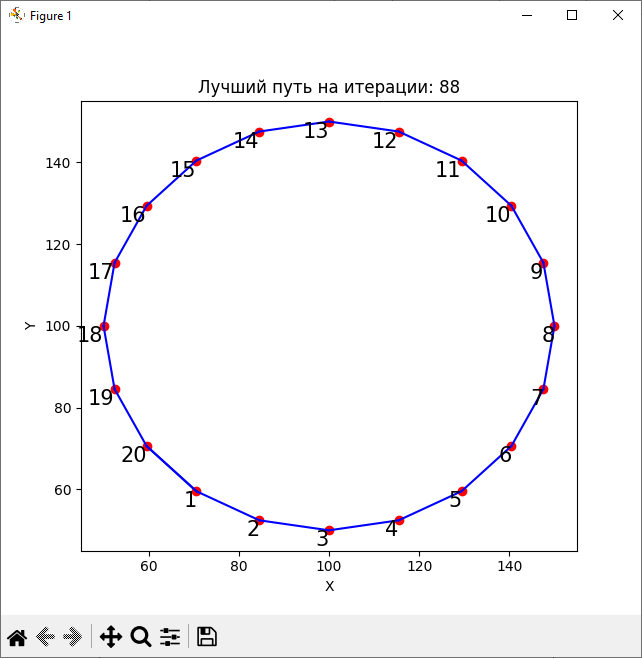


Рисунок 2 – Финал работы алгоритма на третьей итерации.

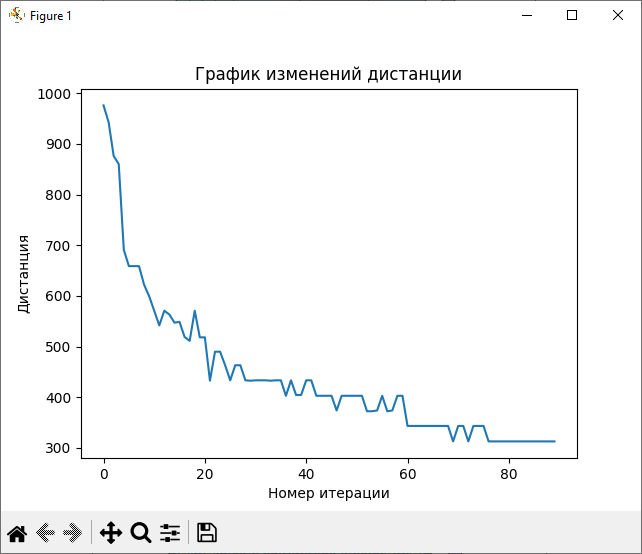


Рисунок 2 – График изменения дистанции работы алгоритма.

Ниже представлена таблица результатов для всех функций с различными параметрами. Количество итераций равнялось 90.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Число городов** | **Размер популяции** | **Вероятность мутации** | **Время выполнения** | **Длина окружности** | **Финальная дистанция** |
| 5 | 40 | 0,01 | 0.0220 | 293.89 | 293.89 |
| 5 | 40 | 0,1 | 0.0220 | 293.89 | 293.89 |
| 5 | 80 | 0,01 | 0.0450 | 293.89 | 293.89 |
| 5 | 80 | 0,1 | 0.0440 | 293.89 | 293.89 |
| 5 | 160 | 0,01 | 0.0891 | 293.89 | 293.89 |
| 5 | 160 | 0,1 | 0.0901 | 293.89 | 293.89 |
| 5 | 320 | 0,01 | 0.1762 | 293.89 | 293.89 |
| 5 | 320 | 0,1 | 0.1772 | 293.89 | 293.89 |
| 10 | 40 | 0,01 | 0.3463 | 309.02 | 451.10 |
| 10 | 40 | 0,1 | 0.3603 | 309.02 | 478.12 |
| 10 | 80 | 0,01 | 0.1161 | 309.02 | 309.02 |
| 10 | 80 | 0,1 | 0.7115 | 309.02 | 364.77 |
| 10 | 160 | 0,01 | 0.2793 | 309.02 | 309.02 |
| 10 | 160 | 0,1 | 1.4096 | 309.02 | 309.02 |
| 10 | 320 | 0,01 | 0.5049 | 309.02 | 309.02 |
| 10 | 320 | 0,1 | 1.1674 | 309.02 | 309.02 |
| 15 | 40 | 0,01 | 0.5115 | 311.87 | 693.19 |
| 15 | 40 | 0,1 | 0.5255 | 311.87 | 677.35 |
| 15 | 80 | 0,01 | 0.8828 | 311.87 | 311.87 |
| 15 | 80 | 0,1 | 1.0159 | 311.87 | 587.62 |
| 15 | 160 | 0,01 | 0.9036 | 311.87 | 311.87 |
| 15 | 160 | 0,1 | 2.1083 | 311.87 | 532.99 |
| 15 | 320 | 0,01 | 1.3553 | 311.87 | 311.87 |
| 15 | 320 | 0,1 | 4.1003 | 311.87 | 549.63 |
| 20 | 40 | 0,01 | 0.6592 | 312.87 | 825.96 |
| 20 | 40 | 0,1 | 0.6828 | 312.87 | 1023.33 |
| 20 | 80 | 0,01 | 1.3581 | 312.87 | 426.33 |
| 20 | 80 | 0,1 | 1.3873 | 312.87 | 926.40 |
| 20 | 160 | 0,01 | 1.8807 | 312.87 | 402.90 |
| 20 | 160 | 0,1 | 2.7953 | 312.87 | 850.64 |
| 20 | 320 | 0,01 | 2.5906 | 312.87 | 312.87 |
| 20 | 320 | 0,1 | 5.4088 | 312.87 | 791.61 |
| 40 | 40 | 0,01 | 1.2860 | 313.84 | 1571.94 |
| 40 | 40 | 0,1 | 1.3164 | 313.84 | 1962.77 |
| 40 | 80 | 0,01 | 2.6114 | 313.84 | 1265.50 |
| 40 | 80 | 0,1 | 2.6191 | 313.84 | 1844.19 |
| 40 | 160 | 0,01 | 5.1563 | 313.84 | 772.87 |
| 40 | 160 | 0,1 | 5.2080 | 313.84 | 1808.39 |
| 40 | 320 | 0,01 | 10.3030 | 313.84 | 741.24 |
| 40 | 320 | 0,1 | 10.4426 | 313.84 | 1741.41 |

Таблица 1 – Результаты работы программы при изменении гиперпараметров

Из представленных результатов можно сделать несколько выводов:

1. Число городов и размер популяции:

- Увеличение числа городов приводит к увеличению времени выполнения и длины окружности.

- Увеличение размера популяции также увеличивает время выполнения, но не всегда ведет к увеличению длины окружности.

2. Вероятность мутации:

- Увеличение вероятности мутации также увеличивает время выполнения, но не всегда ведет к увеличению длины окружности.

- Вероятно, высокая вероятность мутации может привести к быстрой сходимости к локальному оптимуму.

3. Время выполнения:

- Время выполнения значительно растет с увеличением числа городов и размера популяции.

4. Длина окружности:

- Длина окружности является результатом работы алгоритма и зависит от конкретной конфигурации городов, размера популяции и вероятности мутации.

- Иногда длина окружности может оставаться неизменной при изменении параметров алгоритма.

5. Финальная дистанция:

- Финальная дистанция представляет собой длину оптимального маршрута, найденного генетическим алгоритмом.

- Она может оставаться неизменной при изменении параметров алгоритма, если алгоритм сходится к одному и тому же лучшему решению.

6. Анализ:

- При выборе параметров генетического алгоритма необходимо учитывать баланс между временем выполнения и качеством найденного решения.

- Может потребоваться проведение дополнительного анализа и настройка параметров для достижения оптимального результата.

ПРИЛОЖЕНИЕ

Код программы

import math  
  
import numpy as np  
import random  
import operator  
import matplotlib.pyplot as plt  
  
class City:  
 def \_\_init\_\_(self, index: int, max\_x: float, max\_y: float):  
 self.index = index  
 # self.x = random.uniform(0, max\_x)  
 # self.y = random.uniform(0, max\_y)  
 self.x = max\_x  
 self.y = max\_y  
  
 def calculate\_distance(self, other\_city):  
 return np.hypot(self.x - other\_city.x, self.y - other\_city.y)  
  
 def \_\_repr\_\_(self):  
 return f"City(index={self.index}, x={self.x:.2f}, y={self.y:.2f})"  
  
def create\_fixed\_cities():  
 return [  
 City(1, 0),  
 City(0, 1),  
 City(1, 1),  
 City(0, 0),  
 City(0, 2),  
 City(0, 3),  
 ]  
  
def create\_cities(count, max\_x, max\_y):  
 return [City(i, max\_x, max\_y) for i in range(count)]  
  
class Fitness:  
 def \_\_init\_\_(self, route):  
 self.route = route  
 self.distance = 0  
 self.fitness = 0.0  
  
 def route\_distance(self):  
 if self.distance == 0:  
 self.distance = sum(  
 self.route[i].calculate\_distance(self.route[i + 1])  
 for i in range(len(self.route) - 1)  
 ) + self.route[-1].calculate\_distance(self.route[0])  
 return self.distance  
  
 def route\_fitness(self):  
 if self.fitness == 0:  
 self.fitness = 1 / float(self.route\_distance())  
 return self.fitness  
  
def generate\_route(city\_list):  
 return random.sample(city\_list, len(city\_list))  
  
def initial\_population(pop\_size, city\_list):  
 return [generate\_route(city\_list) for \_ in range(pop\_size)]  
  
def rank\_routes(population):  
 fitness\_results = [(i, Fitness(route).route\_fitness(), route) for i, route in enumerate(population)]  
 return sorted(fitness\_results, key=operator.itemgetter(1), reverse=True)  
  
def breed(parent1, parent2):  
 child\_p1 = []  
 gene\_a = int(random.random() \* len(parent1))  
 gene\_b = int(random.random() \* len(parent1))  
 start\_gene, end\_gene = min(gene\_a, gene\_b), max(gene\_a, gene\_b)  
  
 for i in range(start\_gene, end\_gene):  
 child\_p1.append(parent1[i])  
  
 child\_p2 = [gene for gene in parent2 if gene not in child\_p1]  
  
 return child\_p1 + child\_p2  
  
def mutate(individual, mutation\_rate):  
 for swapped in range(len(individual)):  
 if random.random() < mutation\_rate:  
 swap\_with = int(random.random() \* len(individual))  
 individual[swapped], individual[swap\_with] = individual[swap\_with], individual[swapped]  
 return individual  
  
def mutate\_population(population, mutation\_rate):  
 return [mutate(individual, mutation\_rate) for individual in population]  
  
def next\_generation(current\_gen, elite\_size, mutation\_rate):  
 pop\_ranked = rank\_routes(current\_gen)  
 selected\_parents = [pop\_ranked[i][2] for i in range(len(current\_gen) // 4)]  
 next\_gen = []  
  
 next\_gen.extend(current\_gen[:elite\_size])  
  
 while len(next\_gen) < len(current\_gen):  
 parent1, parent2 = random.choices(selected\_parents, k=2)  
 if random.random() < 0.5:  
 child = breed(parent1, parent2)  
 next\_gen.append(child)  
  
 next\_gen = mutate\_population(next\_gen, mutation\_rate)  
 return next\_gen  
  
def plot\_routes(routes, iteration):  
 plt.clf()  
 num\_routes = len(routes)  
  
 for idx, route in enumerate(routes):  
 plt.figure(idx + 1)  
 plt.clf()  
 plt.scatter([city.x for city in route], [city.y for city in route], color='red', edgecolor='black')  
 for i in range(len(route)):  
 plt.plot([route[i - 1].x, route[i].x], [route[i - 1].y, route[i].y], 'b-')  
 plt.text(route[i].x, route[i].y, f"{i + 1}", fontsize=10, ha='right', va='top')  
 plt.plot([route[-1].x, route[0].x], [route[-1].y, route[0].y], 'b-')  
 plt.title(f'Лучший путь {idx + 1} на итерации: {iteration}')  
 plt.xlabel('X')  
 plt.ylabel('Y')  
 plt.pause(0.001)  
  
def plot\_route2(routes, iteration):  
 route = routes[0]  
 plt.clf()  
 plt.scatter([city.x for city in route], [city.y for city in route], color='red')  
 for i in range(len(route)):  
 plt.plot([route[i - 1].x, route[i].x], [route[i - 1].y, route[i].y], 'b-')  
 plt.text(route[i].x, route[i].y, f"{i + 1}", fontsize=15, ha='right', va='top')  
 plt.plot([route[-1].x, route[0].x], [route[-1].y, route[0].y], 'b-')  
 plt.title(f'Лучший путь на итерации: {iteration}')  
 plt.xlabel('X')  
 plt.ylabel('Y')  
 plt.pause(0.4)  
  
def plot\_final\_route(route):  
 plt.figure()  
 plt.scatter([city.x for city in route], [city.y for city in route], color='red')  
 for i in range(len(route)):  
 plt.plot([route[i - 1].x, route[i].x], [route[i - 1].y, route[i].y], 'b-')  
 plt.text(route[i].x, route[i].y, f"{route[i].index}", fontsize=15, ha='right', va='top')  
 plt.plot([route[-1].x, route[0].x], [route[-1].y, route[0].y], 'b-')  
 plt.title('Лучший маршрут')  
 plt.xlabel('X')  
 plt.ylabel('Y')  
 plt.show()  
  
def genetic\_algorithm(city\_list, pop\_size, elite\_size, mutation\_rate, generations, num\_best\_routes=3,  
 convergence\_threshold=10):  
 population = initial\_population(pop\_size, city\_list)  
 print(f"Начальная дистанция: {1 / rank\_routes(population)[0][1]:.2f}")  
 best\_route\_list = []  
 best\_route\_list.append(1 / rank\_routes(population)[0][1])  
 last\_best\_distance = 1 / rank\_routes(population)[0][1]  
 unchanged\_generations = 0  
  
 for i in range(generations):  
 population = next\_generation(population, elite\_size, mutation\_rate)  
 best\_distance = 1 / rank\_routes(population)[0][1]  
 best\_route\_list.append(best\_distance)  
 if best\_distance == last\_best\_distance:  
 unchanged\_generations += 1  
 else:  
 unchanged\_generations = 0  
 last\_best\_distance = best\_distance  
 if unchanged\_generations >= convergence\_threshold:  
 print(f"Достигнут порог сходимости на итерации {i + 1}. Прерываем процесс.")  
 break  
  
 best\_routes\_indices = [rank\_routes(population)[j][0] for j in range(num\_best\_routes)]  
 best\_routes = [population[index] for index in best\_routes\_indices]  
 plot\_route2(best\_routes, i + 1)  
  
 print(f"Финальная дистанция: {best\_distance:.2f}")  
 best\_route\_index = rank\_routes(population)[0][0]  
 best\_route = population[best\_route\_index]  
 return best\_route, best\_route\_list  
  
def create\_triangle\_cities():  
 cities = []  
 side\_length = 100  
 height = (np.sqrt(3)/2) \* side\_length  
  
 vertices = [  
 (0, 0),  
 (side\_length, 0),  
 (side\_length/2, height)  
 ]  
  
 # Adding vertices to cities list  
 for i, (x, y) in enumerate(vertices):  
 cities.append(City(i, x, y))  
  
 # Adding remaining cities along the perimeter of the triangle  
 points\_per\_side = 3  
 for i in range(1, points\_per\_side):  
 # Side 1  
 cities.append(City(len(cities), i \* (side\_length / points\_per\_side), 0))  
 # Side 2  
 cities.append(City(len(cities), side\_length - i \* (side\_length / points\_per\_side), i \* (height / points\_per\_side)))  
 # Side 3  
 cities.append(City(len(cities), (side\_length / 2) - i \* (side\_length / (2 \* points\_per\_side)), height - i \* (height / points\_per\_side)))  
  
 return cities  
  
def create\_circle\_cities(city\_count, radius=50):  
 cities = []  
 t\_array = np.linspace(0, 2 \* math.pi, city\_count, endpoint=False)  
 mid\_x = 100  
 mid\_y = 100  
 dist = 0  
 for i, t in enumerate(t\_array):  
 x = mid\_x + radius \* math.cos(t)  
 y = mid\_y + radius \* math.sin(t)  
 cities.append(City(i, x, y))  
 if len(cities) > 1:  
 dist += cities[-2].calculate\_distance(cities[-1])  
 dist += cities[-1].calculate\_distance(cities[0])  
 return cities,dist  
  
def best\_lenght(city\_list):  
 for city in city\_list:  
 print(city\_list)  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 max\_x, max\_y = 250, 250  
 city\_count = 6  
  
 # city\_list = create\_cities(city\_count, max\_x, max\_y)  
 # city\_list = create\_triangle\_cities()  
 city\_list,dist = create\_circle\_cities(city\_count)  
 print(dist)  
 # print(city\_list[0])  
 plt.ion()  
 best\_route, best\_route\_list = genetic\_algorithm(city\_list=city\_list, pop\_size=40, elite\_size=20,  
 mutation\_rate=0.01, generations=3, num\_best\_routes=3)  
 plt.ioff()  
 plt.show()  
  
 plt.figure()  
 plt.plot(range(len(best\_route\_list)), best\_route\_list)  
 plt.title('График изменений дистанции')  
 plt.xlabel('Номер итерации')  
 plt.ylabel('Дистанция')  
 plt.show()  
  
 best\_route\_indices = [city.index for city in best\_route]  
 print(f"Лучший маршрут: {best\_route\_indices}")  
  
 plot\_final\_route(best\_route)