МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ   
АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ И ПРОГРАММНОЙ ИНЖЕНЕРИИ

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| профессор |  |  |  | Скобцов Ю. А. |
| должность, звание |  | подпись, дата |  | фамилия, инициалы |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3 |
| «Решение задачи коммивояжера с помощью генетических алгоритмов» |
| по дисциплине: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4936 |  |  |  | Нестеренко М.Ю. |
|  |  |  | подпись, дата |  | фамилия, инициалы |

Санкт-Петербург 2022

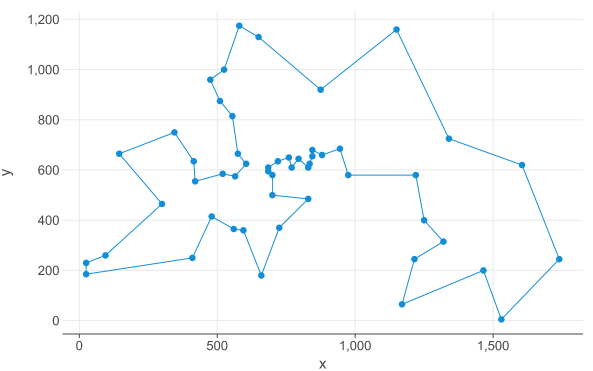
**Цель работы**

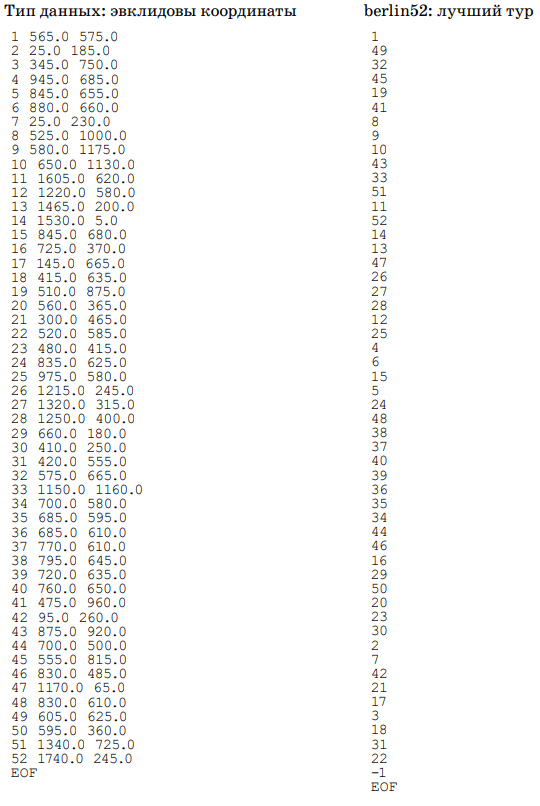
Разработать генетический алгоритм для решения прикладной задачи коммивояжера.

**Вариант задания 14**

Маршрут: Berlin52

Вид представления: Представление порядка





**Теоретические сведения**

Порядковое представление представляет тур как список из n городов; i-й элемент списка – номер от 1 до n-i-1. Идея порядкового представления состоит в следующем. Есть несколько упорядоченных списков городов С, которые служат как точки связи для списков с порядковым представлением. Предположим, для примера, что такой упорядоченный список прост:

(1 2 3 4 5 6 7 8 9)

Тогда тур

1-2-4-3-8-5-9-6-7

будет представлен как список l из ссылок,

l = (1 1 2 1 4 1 3 1 1)

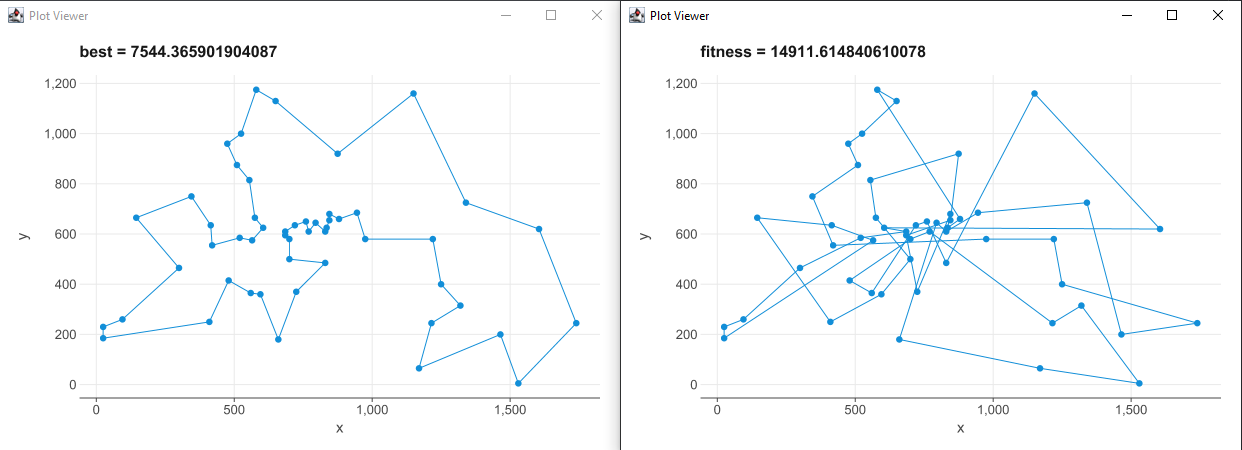
**Код программы**

Код программы написан на языке программирования Kotlin с применением библиотеки lets-plot-kotlin для построения графиков.

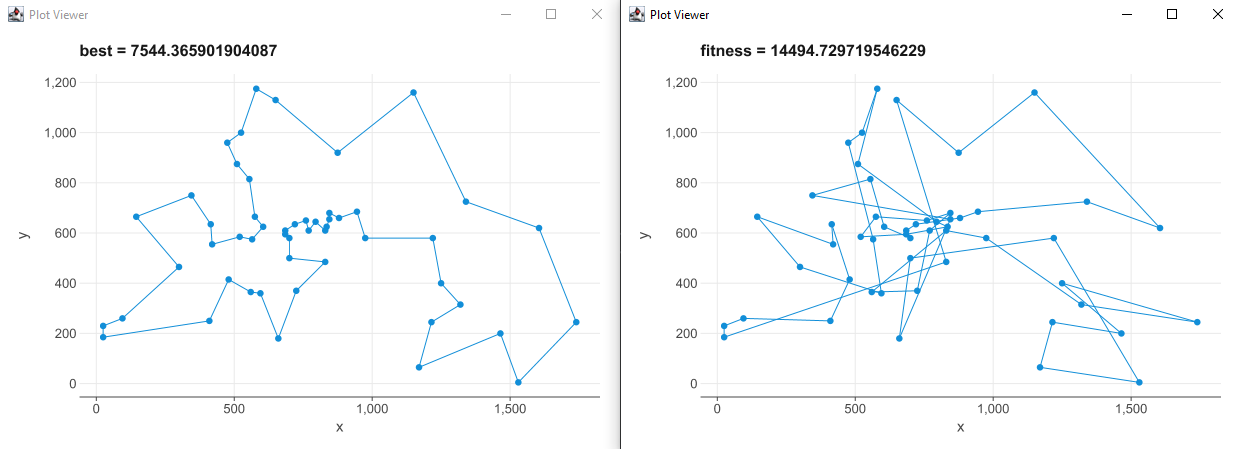
import org.jetbrains.letsPlot.geom.\*  
import org.jetbrains.letsPlot.label.ggtitle  
import org.jetbrains.letsPlot.letsPlot  
import kotlin.math.\*  
import kotlin.random.Random  
  
  
const val *VARIANT* = "#14 \nBerlin52.tsp\n"  
const val *SIZE* = 52  
const val *IS\_MAX\_SEARCHING* = false  
const val *P\_M* = 0.05  
const val *P\_C* = 0.5  
const val *TOURNAMENT\_COUNT* = 2  
const val *INIT\_POPULATION\_COUNT* = 1000  
const val *MAX\_GENERATION\_COUNT* = 100  
  
  
val *population* = *MutableList*(*INIT\_POPULATION\_COUNT*) **{** *mutableListOf*<Int>() **}**val *nodes* = *mutableListOf*(  
 Pair(565, 575), Pair(25, 185), Pair(345, 750), Pair(945, 685),  
 Pair(845, 655), Pair(880, 660), Pair(25, 230), Pair(525, 1000),  
 Pair(580, 1175), Pair(650, 1130), Pair(1605, 620), Pair(1220, 580),  
 Pair(1465, 200), Pair(1530, 5), Pair(845, 680), Pair(725, 370),  
 Pair(145, 665), Pair(415, 635), Pair(510, 875), Pair(560, 365),  
 Pair(300, 465), Pair(520, 585), Pair(480, 415), Pair(835, 625),  
 Pair(975, 580), Pair(1215, 245), Pair(1320, 315), Pair(1250, 400),  
 Pair(660, 180), Pair(410, 250), Pair(420, 555), Pair(575, 665),  
 Pair(1150, 1160), Pair(700, 580), Pair(685, 595), Pair(685, 610),  
 Pair(770, 610), Pair(795, 645), Pair(720, 635), Pair(760, 650),  
 Pair(475, 960), Pair(95, 260), Pair(875, 920), Pair(700, 500),  
 Pair(555, 815), Pair(830, 485), Pair(1170, 65), Pair(830, 610),  
 Pair(605, 625), Pair(595, 360), Pair(1340, 725), Pair(1740, 245),  
)  
val *best* = *mutableListOf*(  
 0, 48, 31, 44, 18, 40, 7, 8, 9, 42, 32, 50, 10,  
 51, 13, 12, 46, 25, 26, 27, 11, 24, 3, 5, 14, 4,  
 23, 47, 37, 36, 39, 38, 35, 34, 33, 43, 45, 15, 28,  
 49, 19, 22, 29, 1, 6, 41, 20, 16, 2, 17, 30, 21  
)  
var *bestCpy* = *best*.*toMutableList*()  
val *ordinalBest* = *MutableList*(*best*.size) **{** i **->** *bestCpy* = *bestCpy*.*map* **{** if (**it** > *bestCpy*[i]) **it** - 1 else **it }**.*toMutableList*()  
 *bestCpy*[i]  
**}**fun main() {  
 val begin: Long = System.currentTimeMillis()  
  
 *println*(*VARIANT*)  
 *population*.*apply* **{** *initPopulation*()  
 *println*("extreme fitness before = $*extremeFitness*")  
 *evolution*()  
 *println*("extreme fitness after = $*extremeFitness*")  
 *println*("best fitness = ${ *ordinalBest*.*fitness* }")  
 *ordinalBest*.*draw*("best = ${*ordinalBest*.*fitness*}")  
 *extremeIndividual*.*draw*("fitness = $*extremeFitness*")  
 **}** val end: Long = System.currentTimeMillis()  
 *println*("Time spent: ${ end - begin } ms")  
}  
  
fun MutableList<Int>.draw(title: String) {  
 val xs = *List*(*SIZE*) **{** *nodes*[**it**].*x* **}** val ys = *List*(*SIZE*) **{** *nodes*[**it**].*y* **}** val data = *mapOf* ("x" *to* xs, "y" *to* ys)  
  
 val copy = *nodes*.*toMutableList*()  
 var p = *letsPlot*(data)  
  
 val firstNode = copy[this[0]]  
 var lastNode = copy[this[0]]  
 for (i in 0..copy.size - 2) {  
 val dx = *mutableListOf*(copy[this[i]].*x*)  
 val dy = *mutableListOf*(copy[this[i]].*y*)  
 copy.removeAt(this[i])  
 dx.add(copy[this[i + 1]].*x*)  
 dy.add(copy[this[i + 1]].*y*)  
 lastNode = copy[this[i + 1]]  
  
 p += geomLine **{** x = dx; y = dy **}** }  
 p += geomLine **{** x = *mutableListOf*(lastNode.*x*, firstNode.*x*); y = *mutableListOf*(lastNode.*y*, firstNode.*y*) **}** p += geomPoint(size = 3) **{** x = "x"; y = "y" **}** + *ggtitle*(title)  
 p.show()  
}  
  
fun MutableList<MutableList<Int>>.initPopulation() {  
 *repeat*(*INIT\_POPULATION\_COUNT*) **{** way **->** */\*val copy = nodes.toMutableList()  
 this[way].add((0 until SIZE).random())  
 var cur = copy[this[way].last()]  
 copy.removeAt(this[way].last())  
  
 repeat(SIZE - 1) { i ->  
 val pull = MutableList(SIZE / 2) { copy[(0 until SIZE - i - 1).random()] }  
 val node2 = if (!IS\_MAX\_SEARCHING) pull.minByOrNull { euclideanDistance(cur, it) }!! else pull.maxByOrNull { euclideanDistance(cur, it) }!!  
 val k = copy.indexOf(node2)  
 cur = node2  
 this[way].add(k)  
 copy.removeAt(k)  
 }\*/  
  
 repeat*(*SIZE*) **{** this[way].add((0 *until SIZE* - **it**).*random*() ) **}  
 }**}  
  
fun MutableList<MutableList<Int>>.evolution() {  
 var generationCounter = 0  
 while (generationCounter++ <= *MAX\_GENERATION\_COUNT /\*&& extremeFitness <= ordinalBest.fitness\*/*) { *// todo* val offspring = *selection*().*apply* **{** *crossing*()  
 *mutation*()  
 **}** *population*.clear()  
 *population*.addAll(offspring)  
 *println*("distinct = ${*distinct*().size}")  
 }  
 *println*("GENERATION COUNT = ${generationCounter - 2}")  
}  
  
fun MutableList<MutableList<Int>>.selection(): MutableList<MutableList<Int>> {  
 val newPopulation = *mutableListOf*(*extremeIndividual*)  
 *repeat*(*INIT\_POPULATION\_COUNT* - 1) **{** var ies = *List*(*TOURNAMENT\_COUNT*) **{** (0 *until INIT\_POPULATION\_COUNT*).*random*() **}** while (ies.size != ies.*distinct*().size) {  
 ies = *List*(*TOURNAMENT\_COUNT*) **{** (0 *until INIT\_POPULATION\_COUNT*).*random*() **}** }  
 val tournamentIndividuals = *MutableList*(*TOURNAMENT\_COUNT*) **{** this[ies[**it**]] **}** newPopulation.add(tournamentIndividuals.*extremeIndividual*)  
 **}** return newPopulation  
}  
  
fun MutableList<MutableList<Int>>.crossing() {  
 for (i in 0 *until INIT\_POPULATION\_COUNT step* 2) {  
 if (Random.nextDouble() <= *P\_C*) {  
 val k = (0 *until SIZE*).*random*()  
 val firstPart = this[i].subList(k, *SIZE*)  
 val secondPart = this[i + 1].subList(k, *SIZE*)  
  
 this[i] = (this[i].subList(0, k) + secondPart).*toMutableList*()  
 this[i + 1] = (this[i + 1].subList(0, k) + firstPart).*toMutableList*()  
 }  
 }  
}  
  
fun MutableList<MutableList<Int>>.mutation() {  
 *repeat*(*INIT\_POPULATION\_COUNT*) **{** i **->** if (Random.nextDouble() <= *P\_M*) {  
 val k = (0 *until SIZE*).*random*()  
 this[i][k] = (0 *until SIZE* - k).*random*()  
 }  
 **}**}  
  
  
val MutableList<MutableList<Int>>.*extremeFitness* get () = if (*IS\_MAX\_SEARCHING*) *maxFitness* else *minFitness*val MutableList<MutableList<Int>>.*extremeIndividual* get () = if (*IS\_MAX\_SEARCHING*) *maxByOrNull* **{ it**.*fitness* **}**!! else *minByOrNull* **{ it**.*fitness* **}**!!  
val MutableList<MutableList<Int>>.*maxFitness* get () = this.*maxOf* **{ it**.*fitness* **}**val MutableList<MutableList<Int>>.*minFitness* get () = this.*minOf* **{ it**.*fitness* **}**val Pair<Int, Int>.*x* get() = first  
val Pair<Int, Int>.*y* get() = second  
  
val MutableList<Int>.*fitness*: Double  
 get() = *fitness*()  
  
fun MutableList<Int>.fitness(): Double {  
 var fitness = 0.0  
 val copy = *nodes*.*toMutableList*()  
 val firstNode = copy[this[0]]  
 var lastNode = copy[this[0]]  
  
 for (i in 0..size - 2) {  
 val from = copy[this[i]]  
 copy.removeAt(this[i])  
 val to = copy[this[i + 1]]  
 lastNode = to  
  
 fitness += *euclideanDistance*(from, to)  
 }  
 fitness += *euclideanDistance*(firstNode, lastNode)  
 return fitness  
}  
  
fun euclideanDistance(from: Pair<Int, Int>, to: Pair<Int, Int>) =  
 *sqrt*((from.*x* - to.*x*).toDouble().*pow*(2) + (from.*y* - to.*y*).toDouble().*pow*(2))

**Результат выполнения**

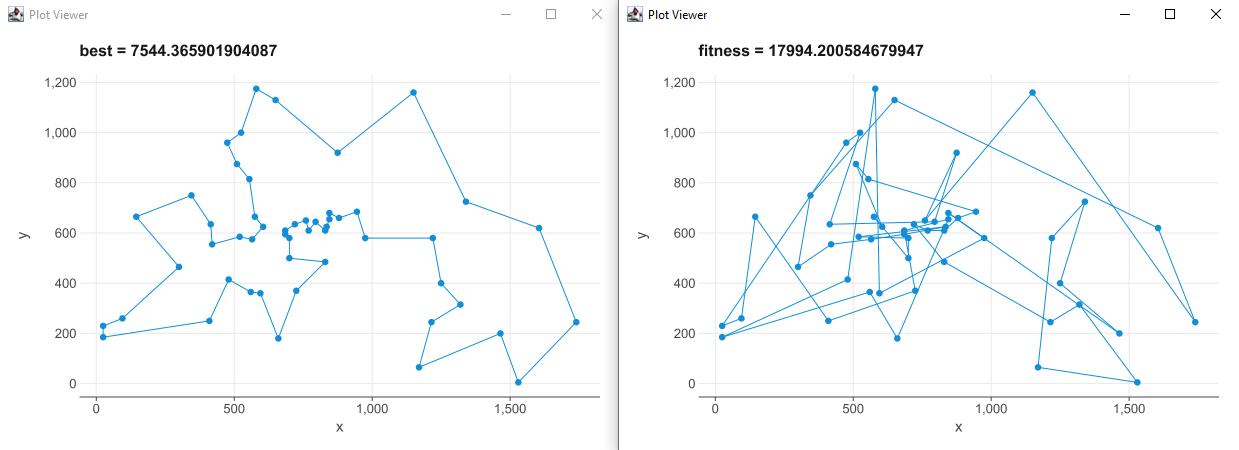
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pc | Pm | Nпок | Nпоп | Nтурн |
| 0,5 | 0,05 | 100 | 1000 | 2 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pc | Pm | Nпок | Nпоп | Nтурн |
| 0,5 | 0,1 | 100 | 1000 | 2 |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Pc | Pm | Nпок | Nпоп | Nтурн |
| 0,25 | 0,2 | 100 | 100 | 2 |



**Зависимость работы ГА от параметров**

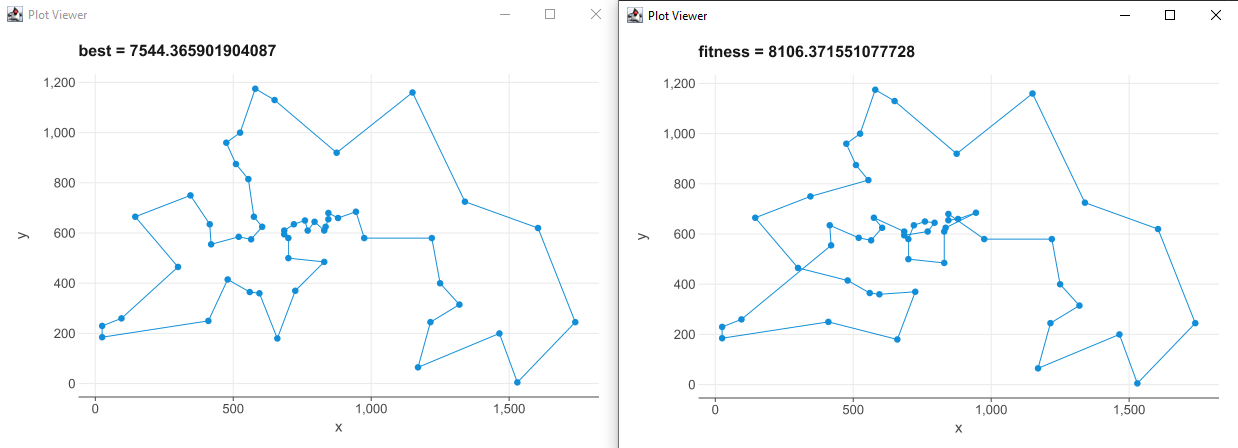
При увеличении вероятности кроссинговера (Pc) точность алгоритма в среднем возрастает. Несмотря на классический алгоритм скрещивания в данном представлении, оно в разы улучшает результаты выполнения.

При увеличении вероятности мутации (Pm) точность алгоритма в среднем падает. Поскольку при слишком частых мутациях, удачные решения не могут устояться и теряются. Самым удачным оказалось значение Pm = 0.25.

При увеличении размера популяции (N) точность алгоритма в среднем растет. Так как изначальная большая выборка позволяет выбрать наиболее удачные решения и модифицировать их.

**Эвристика**

Дополним ранее созданный алгоритм эвристическими методами, позволяющими задать генерацию изначальной популяции более благоприятными для нахождения оптимального результата.



Результат улучшает почти в два раза. Поскольку классический алгоритм кроссинговера не всегда способен провести оптимальное скрещивания, то необходимо применять дополнительный эвристические методы.

**Контрольный вопрос**

В чем основная идея применения ГА для решения задачи коммивояжера?

Поскольку задача коммивояжера имеет алгоритмическую сложность O(n!), то нахождение наилучшего пути методом перебора при больших размерах почти невозможна для современных компьютеров. Поэтому для решения используют такие приближенные алгоритмы, как, например, генетические. Такие алгоритмы частично используют методы перебора, оптимизируя их для уменьшения времени нахождения решения близкого к оптимальному.

**Вывод**

В ходе выполнения лабораторной работы были получены основные навыки разработки и модификации генетического алгоритма для решения прикладных алгоритмических задач.