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 №4 |
| «Генетическое программирование» |
| по дисциплине: ЭВОЛЮЦИОННЫЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНО-ИНФОРМАЦИОННЫХ СИСТЕМ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

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

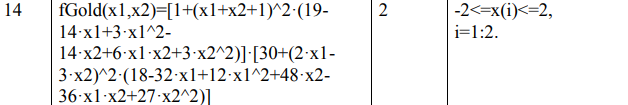
Санкт-Петербург 2022

**Цель работы**

Разработать генетический алгоритм для решения прикладной задачи коммивояжера.

**Вариант задания 14**

Вид представления: Древовидное



**Теоретические сведения**

В генетическом программировании (ГП) в качестве особи выступает программа, представленная в определенном формате, которая решает некоторую задачу. Часто это выполняется с использованием обучающих данных и индуктивного вывода. ГП очень близко к машинному обучению и поэтому в качестве фитнесс-функции как правило выступают функции ошибки. ГП работает с генетическим материалом переменной длины, что требует нестандартной формы представления генома и соответствующих генетических операторов.

Программы составляются из переменных, констант и функций, которые связаны некоторыми синтаксическими правилами. Поэтому определяется терминальное множество, содержащее константы и переменные, и функциональное множество, которое состоит, прежде всего, из операторов и необходимых элементарных функций (exp(x), sin(x) и т.п.). Следует отметить, что терминалы и функции играют различную роль. Терминалы обеспечивают входные значения в систему (программу), в то время как функции используются при обработке значений внутри системы. Термины «функции» и «терминалы» взяты из древовидного представления, и соответствуют узлам древовидных (или графоподобных) структур.

**Код программы**

Код программы, написанный на языке программирования Kotlin:

**Main.kt**  
fun main() {  
 GA().evolve()  
}

**Constants.kt**  
object Constants {  
  
 const val FROM = -2.0  
 const val TO = 2.0  
 const val P\_M = 0.05  
 const val P\_C = 0.5  
 const val TOURNAMENT\_SIZE = 2  
 const val INIT\_POPULATION\_SIZE = 100  
 const val REROLL\_COUNT = 5  
 const val MAX\_DEPTH = 6  
 const val MAX\_GENERATION\_COUNT = 100  
  
}

**Context.kt**  
import kotlin.math.pow  
  
object Context {  
val targetFunction: (Map<String, Double>) -> Double = **{** vars **->** (1 + (vars["x1"]!! + vars["x2"]!! + 1).*pow*(2) \* (19 - 14 \* vars["x1"]!! + 3 \* vars["x1"]!!.*pow*(2) - 14 \* vars["x2"]!! + 6 \* vars["x1"]!! \* vars["x2"]!! + 3 \* vars["x2"]!!.*pow*(2))) \*  
 (30 + (2 \* vars["x1"]!! - 3 \* vars["x2"]!!).*pow*(2) \* (18 - 32 \* vars["x1"]!! + 12 \* vars["x1"]!!.*pow*(2) + 48 \* vars["x2"]!! - 36 \* vars["x1"]!! \* vars["x2"]!! + 27 \* vars["x2"]!!.*pow*(2)))  
 **}**val constants = *listOf*(  
 1.0, 1.0, 2.0, 19.0, 14.0, 3.0, 2.0,  
 14.0, 6.0, 3.0, 2.0, 30.0, 2.0, 18.0,  
 32.0, 12.0, 2.0, 48.0, 36.0, 27.0, 2.0  
 )  
  
 val vars = *mutableMapOf*(  
 "x1" *to* 0.0,  
 "x2" *to* 0.0,)  
}

**Expression.kt**  
import Constants.*FROM*import Constants.*MAX\_DEPTH*import Constants.*P\_C*import Constants.*P\_M*import Constants.*TO*import Context.*vars*import Context.*constants*import Context.*targetFunction*import kotlin.math.\*  
import kotlin.random.Random  
  
class Expression() {  
  
 constructor(f: Function): this() {  
 current = f  
 }  
 fun clone(): Expression {  
 val cloned = Expression(current!!)  
 cloned.left = left?.clone()  
 cloned.right = right?.clone()  
 return cloned  
 }  
  
 private var current: Function? = null  
 private var left: Expression? = null  
 private var right: Expression? = null  
  
 private val isTerminal get() = left == null && right == null  
 private val isNotTerminal get() = !isTerminal  
  
 private val result: Double get() =  
 if (isNotTerminal)  
 when(current) {  
 Function.Sum -> left!!.result + right!!.result  
 Function.Sub -> left!!.result - right!!.result  
 Function.Mul -> left!!.result \* right!!.result  
 Function.Div -> *with* (left!!.result / right!!.result) **{** if (this.*isInfinite*()) Double.MAX\_VALUE else this **}** Function.Pow -> *abs*(left!!.result).*pow*(*abs*(right!!.result))  
 Function.Sin -> *sin*(left!!.result)  
 Function.Cos -> *cos*(left!!.result)  
 Function.Abs -> *abs*(left!!.result)  
 Function.Exp -> *exp*(left!!.result)  
 else -> 0.0  
 }  
 else  
 when(current) {  
 is Function.Const -> (current as Function.Const).get  
 is Function.Var -> (current as Function.Var).get  
 else -> 0.0  
 }  
  
 fun generate(depth: Int = 0) {  
 if (depth == *MAX\_DEPTH*) {  
 current = randomTerminal  
 } else {  
 if ((0..1).*random*() == 0 && depth > 2) {  
 current = randomTerminal  
 } else {  
 current = randomFunction  
 left = Expression().*apply* **{** generate(depth + 1) **}** if (!current!!.isUnary)  
 right = Expression().*apply* **{** generate(depth + 1) **}** }  
 }  
 }  
  
 private fun cut(depth: Int = 0) {  
 if (depth > *MAX\_DEPTH*) {  
 killChilds()  
 } else {  
 left?.cut(depth + 1)  
 right?.cut(depth + 1)  
 }  
 }  
  
 private fun killChilds() {  
 current = when(current) {  
 is Function.Const,  
 is Function.Var -> current  
 else -> randomTerminal  
 }  
 left = null  
 right = null  
 }  
  
 fun mutation() {  
 if (Random.nextDouble() <= *P\_M*) {  
 when ((0..3).*random*()) {  
 *// растущая* 0 -> {  
 randomNode.*apply* **{** current = randomFunction  
 left = Expression().*apply* **{** generate((1..*MAX\_DEPTH*).*random*()) **}** right = Expression().*apply* **{** generate((1..*MAX\_DEPTH*).*random*()) **}  
 }** }  
 *// обмен* 1 -> {  
 randomNode.*apply* **{** if (current!!.isUnary)  
 left = Expression().*apply* **{** generate((1..*MAX\_DEPTH*).*random*()) **}** else  
 left = right.*also* **{** right = left **}  
 }** }  
 *// подмена корня* 2 -> {  
 val old = this.clone()  
 current = randomFunction  
 left = old  
 right = if (current!!.isUnary) null  
 else Expression().*apply* **{** generate(1) **}** }  
 *// смена функции* else -> {  
 randomNode.*apply* **{** val old = this.current  
 this.current = randomFunction  
  
 if (old!!.isUnary && !this.current!!.isUnary) {  
 this.right = Expression().*apply* **{** current = randomTerminal **}** }  
 else if (this.current!!.isUnary)  
 this.right = null  
 **}** }  
 }  
 cut()  
 }  
 }  
  
 fun crossing(recessive: Expression) {  
 if (Random.nextDouble() <= *P\_C*) {  
 randomNode.*apply* **{** val recessiveRandomNode = recessive.randomNode.clone()  
 this.current = recessiveRandomNode.current  
 this.left = recessiveRandomNode.left  
 this.right = recessiveRandomNode.right  
 **}** recessive.randomNode.*apply* **{** val dominantRandomNode = this@Expression.randomNode.clone()  
 this.current = dominantRandomNode.current  
 this.left = dominantRandomNode.left  
 this.right = dominantRandomNode.right  
 **}** cut()  
 recessive.cut()  
 }  
 }  
  
 private val randomTerminal get() = if (Random.nextDouble() > 0.5) randomConst else randomVariable  
 private val randomVariable get() = Function.Var(*vars*.keys.*random*())  
 private val randomConst get() = Function.Const(*constants*.*random*())  
 private val randomUnary get() = when((0..3).*random*()) {  
 0 -> Function.Abs  
 1 -> Function.Cos  
 2 -> Function.Sin  
 3 -> Function.Exp  
 else -> Function.Cos  
 }  
 private val randomFunction get() = when((0..8).*random*()) {  
 0 -> Function.Sum  
 1 -> Function.Sub  
 2 -> Function.Mul  
 3 -> Function.Div  
 4 -> Function.Abs  
 5 -> Function.Cos  
 6 -> Function.Sin  
 7 -> Function.Exp  
 8 -> Function.Pow  
 else -> Function.Sum  
 }  
 private val randomNode get(): Expression = when ((0..2).*random*()) {  
 0 -> this  
 1 ->  
 if (left?.isTerminal != false)  
 this  
 else  
 left!!.randomNode  
 2 ->  
 if (current!!.isUnary) {  
 if (left?.isTerminal != false)  
 this  
 else  
 left!!.randomNode  
 }  
 else {  
 if (right?.isTerminal != false)  
 this  
 else  
 right!!.randomNode  
 }  
 else -> this  
 }  
 val fitness get(): Double {  
 var dist = 0.0  
 *vars*.keys.*forEach* **{** *vars*[**it**] = *FROM* **}** *repeat*(1) **{** *vars*.keys.*forEach* **{** *vars*[**it**] = *vars*[**it**]!! + (*abs*(*FROM*) + *abs*(*TO*)) / 1 **}** dist += *abs*(result - *targetFunction*(*vars*))  
 **}** return dist  
 }  
  
 override fun toString() =  
 if (current!!.isUnary)  
 "$current(${left ?: ""})".*trim*()  
 else  
 "${left ?: ""} $current ${right ?: ""}".*trim*()  
}

**Function.kt**  
import Context.*vars*sealed class Function(  
 val value: String,  
 val isUnary: Boolean = false,  
) {  
 override fun toString() = value  
  
 object Sum: Function(  
 value = "+",  
 )  
  
 object Sub: Function(  
 value = "-",  
 )  
  
 object Mul: Function(  
 value = "\*",  
 )  
  
 object Div: Function(  
 value = "/",  
 )  
  
 object Abs: Function(  
 value = "abs",  
 isUnary = true  
 )  
  
 object Cos: Function(  
 value = "cos",  
 isUnary = true  
 )  
  
 object Sin: Function(  
 value = "sin",  
 isUnary = true  
 )  
  
 object Exp: Function(  
 value = "exp",  
 isUnary = true  
 )  
  
 object Pow: Function(  
 value = "^"  
 )  
  
 data class Var(val name: String): Function(  
 value = name  
 ) {  
 override fun toString() = value  
 val get get() = *vars*[value]!!  
 }  
  
 data class Const(val const: Double): Function(  
 value = const.toString(),  
 ) {  
 override fun toString() = value  
 val get get() = value.*toDouble*()  
 }  
}

**GA.kt**  
import Constants.*INIT\_POPULATION\_SIZE*import Constants.*MAX\_GENERATION\_COUNT*import Constants.*REROLL\_COUNT*import Constants.*TOURNAMENT\_SIZE*class GA {  
  
 private val population = *MutableList*(*INIT\_POPULATION\_SIZE*) **{** Expression().*apply* **{** generate() **} }** private val MutableList<Expression>.*bestIndividual* get() = *minBy* **{ it**.fitness **}** private val MutableList<Expression>.*worstIndividual* get() = *maxBy* **{ it**.fitness **}** private fun selection() {  
 val newPopulation = *mutableListOf*(population.*bestIndividual*)  
 *repeat*(*INIT\_POPULATION\_SIZE* - 1) **{** i **->** var ies = *List*(*TOURNAMENT\_SIZE*) **{** (0 *until INIT\_POPULATION\_SIZE*).*random*() **}** while (ies.size != ies.*distinct*().size) {  
 ies = *List*(*TOURNAMENT\_SIZE*) **{** (0 *until INIT\_POPULATION\_SIZE*).*random*() **}** }  
 val tournamentIndividuals = *MutableList*(*TOURNAMENT\_SIZE*) **{** population[ies[**it**]] **}** newPopulation.add(tournamentIndividuals.*bestIndividual*.clone())  
 **}** population.clear()  
 population.addAll(newPopulation)  
 }  
  
 private fun crossing() {  
 for (i in 0 *until INIT\_POPULATION\_SIZE step* 2) {  
 population[i].crossing(population[i + 1])  
 }  
 }  
  
 private fun mutation() {  
 *repeat*(*INIT\_POPULATION\_SIZE*) **{** population[**it**].mutation()  
 **}** }  
  
 private fun reroll() {  
 val best = population.*bestIndividual* population.remove(best)  
 *repeat*(*REROLL\_COUNT*) **{** val i = (0 *until* population.size).*random*()  
 population.removeAt(i)  
 population.add(Expression().*apply* **{** generate() **}**)  
 **}** population.add(best)  
 }  
  
 fun evolve() {  
 *println*("best before = ${population.*bestIndividual*}")  
 *println*("fit = ${population.*bestIndividual*.fitness}\n")  
 var counter = 0  
 val begin = System.currentTimeMillis()  
 while (++counter <= *MAX\_GENERATION\_COUNT* || population.*bestIndividual*.fitness > 1.0) {  
 selection()  
 crossing()  
 mutation()  
 reroll()  
 }  
 val end = System.currentTimeMillis()  
 *println*("best after = ${population.*bestIndividual*}")  
 *println*("fit = ${population.*bestIndividual*.fitness}\n")  
 *println*("Time spent: ${ end - begin } ms")  
 }  
  
}

**Результат выполнения**

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

Значение фитнесс функции: 0,09

Время выполнения: 190 мс

x1 \* abs(x1 \* x1 ^ x1 + x1 / exp(x2) / 48.0 + 18.0) + abs(x2 - x1) + abs(x1 \* 27.0 - x2 ^ abs(x2) + x2 - x1 / exp(abs(x2)) / abs(14.0 ^ x2) + 18.0)

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

Значение фитнесс функции: 0,3

Время выполнения: 140 мс

sin(cos(x1 - 3.0)) - cos(abs(exp(x2) ^ x1 + 3.0)) - exp(1.0) \* abs(x1 - 3.0 / 30.0 + 1.0 + sin(abs(x2)) \* abs(exp(x2) ^ x1 + 3.0) - sin(cos(2.0)) - abs(x1 \* 30.0))

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Pc | Pm | Nпок | Nпоп | Nтурн | Nобнов |
| 0,25 | 0,05 | 100 | 1000 | 2 | 10 |

Значение фитнесс функции: 0,002

Время выполнения: 632 мс

abs(abs(x2 / 12.0 / sin(x1) \* 2.0 \* 12.0 \* x2 / x2) \* abs(abs(2.0) \* 2.0 \* 3.0) \* 2.0 \* 14.0 - 2.0 / x1 + 48.0 \* abs(abs(12.0 + 2.0) \* abs(x1 \* x1)) \* x1 \* 27.0)

**Зависимость работы ГА от параметров**

При очень больших и малых значениях вероятность кроссинговера (Pc) генетический алгоритм резко увеличивает погрешность вычислений. Оптимальными значениями являются 0.5 – 0.05.

При уменьшении вероятности мутации (Pm) точность алгоритма линейно растет. Это объясняется тем, что популяция успевает устояться и эволюционировать.

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

При уменьшении количество обновляемых особей (Nобнов) точность алгоритма растет. Алгоритм удаляет Nобнов случайных особей в популяции и заменяет их новыми для внесения разнообразия в генотип.

При уменьшении количество обновляемых особей (Nобнов худш) точность алгоритма в среднем растет. Алгоритм удаляет Nобнов худш худших особей в популяции и заменяет их новыми для внесения разнообразия в генотип. В отличии от удаления случайных особей, позволяет получать более точные результаты.

**Контрольный вопрос**

Что такое интроны?

Интрон – ненужный участок кода (функции), который не изменяет хода вычислений алгоритма. Например, минус на минус, отрицание отрицания, модуль модуля и так далее.

**Вывод**

В ходе выполнения лабораторной работы были получены основные навыки разработки и модификации алгоритма генетического программирования на высокоуровневом языке программирования.