Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение высшего образования

**ИРКУТСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

|  |
| --- |
| Институт Высоких Технологий |
| наименование института |
| Кафедра автоматизированных систем |
| наименование кафедры |

**Лабораторная работа**

по дисциплине:

|  |
| --- |
| **Технологии искусственного интеллекта** |
| Программирование генетического алгоритма |

(Наименование темы)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Выполнил студент группы: | ИСТбп-16 |  |  |  | Литвинов В.А. |
|  |  |  |  |  | Фамилия Имя Отчество |
|  |  |  |  |  |  |
| Проверил: |  |  |  |  | Пестерев Д.В. |
|  | должность |  | подпись |  | Фамилия Имя Отчество |

Иркутск 2020 г

ОГЛАВЛЕНИЕ

[ВВЕДЕНИЕ 3](#_Toc26534345)

[1 Постановка задачи 6](#_Toc26534346)

[Необходимо реализовать генетический алгоритм, для этого требуется: 6](#_Toc26534347)

[2 Результат работы программы 7](#_Toc26534348)

[3 Листинг реализации генетического алгоритма 8](#_Toc26534349)

[ЗАКЛЮЧЕНИЕ 12](#_Toc26534350)

ВВЕДЕНИЕ

Генетический алгоритм (англ. genetic algorithm, ГА) — это эвристический алгоритм поиска, используемый для решения задач оптимизации и моделирования путѐм случайного подбора, комбинирования и вариации искомых параметров с использованием механизмов, аналогичных естественному отбору в природе. Является разновидностью эволюционных вычислений, с помощью которых решаются оптимизационные задачи с использованием методов естественной эволюции, таких как наследование, мутации, отбор и кроссинговер. Отличительной особенностью генетического алгоритма является акцент на использование оператора «скрещивания», который производит операцию рекомбинации решений-кандидатов, роль которой аналогична роли скрещивания в живой природе.

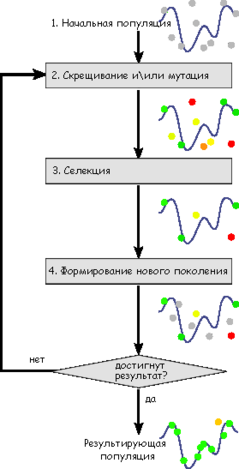


Рис. Схема работы генетического алгоритма

Задача формализуется таким образом, чтобы еѐ решение могло быть закодировано в виде вектора («генотипа») генов, где каждый ген может быть битом, числом или неким другим объектом. В классических реализациях ГА предполагается, что генотип имеет фиксированную длину. Однако существуют вариации ГА, свободные от этого ограничения.

Некоторым, обычно случайным, образом создаѐтся множество генотипов начальной популяции. Они оцениваются с использованием «функции приспособленности», в результате чего с каждым генотипом ассоциируется определѐнное значение («приспособленность»), которое определяет насколько хорошо фенотип, им описываемый, решает поставленную задачу.

При выборе «функции приспособленности» (или fitness function в англоязычной литературе) важно следить, чтобы еѐ «рельеф» был «гладким».

Из полученного множества решений («поколения») с учѐтом значения «приспособленности» выбираются решения (обычно лучшие особи имеют большую вероятность быть выбранными), к которым применяются «генетические операторы» (в большинстве случаев «скрещивание» — crossover и «мутация» — mutation), результатом чего является получение новых решений. Для них также вычисляется значение приспособленности, и затем производится отбор («селекция») лучших решений в следующее поколение. Этот набор действий повторяется итеративно, так моделируется «эволюционный процесс», продолжающийся несколько жизненных циклов (поколений), пока не будет выполнен критерий остановки алгоритма. Таким критерием может быть:

* нахождение глобального, либо субоптимального решения;
* исчерпание числа поколений, отпущенных на эволюцию;
* исчерпание времени, отпущенного на эволюцию.

Таким образом, можно выделить следующие этапы генетического алгоритма:

1. Задать целевую функцию (приспособленности) для особей популяции

2. Создать начальную популяцию

3. (Начало цикла)

4. Размножение (скрещивание)

5. Мутирование

6. Вычислить значение целевой функции для всех особей

7. Формирование нового поколения (селекция)

8. Если выполняются условия остановки, то (конец цикла), иначе (начало цикла).

**Создание начальной популяции**

Перед первым шагом нужно случайным образом создать начальную популяцию; даже если она окажется совершенно неконкурентоспособной, вероятно, что генетический алгоритм все равно достаточно быстро переведет еѐ в жизнеспособную популяцию. Таким образом, на первом шаге можно особенно не стараться сделать слишком уж приспособленных особей, достаточно, чтобы они соответствовали формату особей популяции, и на них можно было подсчитать функцию приспособленности (Fitness). Итогом первого шага является популяция H, состоящая из N особей.

**Размножение (Скрещивание)**

Размножение в генетических алгоритмах обычно половое — чтобы произвести потомка, нужны несколько родителей, обычно два.

Размножение в разных алгоритмах определяется по-разному — оно, конечно, зависит от представления данных. Главное требование к размножению — чтобы потомок или потомки имели возможность унаследовать черты обоих родителей, «смешав» их каким-либо способом.

Почему особи для размножения обычно выбираются из всей популяции H, а не из выживших на первом шаге элементов H0 (хотя последний вариант тоже имеет право на существование)? Дело в том, что главный бич многих генетических алгоритмов — недостаток разнообразия (diversity) в особях. Достаточно быстро выделяется один-единственный генотип, который представляет собой локальный максимум, а затем все элементы популяции проигрывают ему отбор, и вся популяция «забивается» копиями этой особи. Есть разные способы борьбы с таким нежелательным эффектом; один из них — выбор для размножения не самых приспособленных, но вообще всех особей.

**Мутации**

К мутациям относится все то же самое, что и к размножению: есть некоторая доля мутантов m, являющаяся параметром генетического алгоритма, и на шаге мутаций нужно выбрать mN особей, а затем изменить их в соответствии с заранее определѐнными операциями мутации.

**Отбор**

На этапе отбора нужно из всей популяции выбрать определѐнную еѐ долю, которая останется «в живых» на этом этапе эволюции. Есть разные способы проводить отбор. Вероятность выживания особи h должна зависеть от значения функции приспособленности Fitness(h). Сама доля выживших s обычно является параметром генетического алгоритма, и еѐ просто задают заранее. По итогам отбора из N особей популяции H должны остаться sN особей, которые войдут в итоговую популяцию H'. Остальные особи погибают.

1. Постановка задачи

Необходимо реализовать генетический алгоритм, для этого требуется:

* Создать искомую функцию на участке [0..10], согласно варианту.
* Создать популяцию из 20 особей, со ста признаками, соответствующими отрезку [0..10] (интервал между признаками = 0,1).
* Создать алгоритм, размножения и мутации, способный на каждом шаге удваивать популяцию.
* Уничтожать половину удвоенной популяции, эвклидово расстояние признаков которых максимально удалено от искомой функции.
* Остановить мутационный процесс при достижении эвклидова расстояния = 0,5.
* Вывести признаки особи с наименьшим эвклидовым расстоянием в виде графика и рядом график функции для сравнения.

|  |  |
| --- | --- |
| Вариант **№** | **Функция** |
| 3 | Y=x2 |

1. Результат работы программы

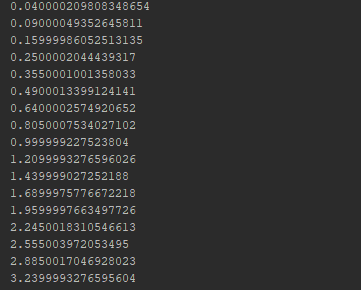


Рисунок 1 – Набор характеристик лучшей особи

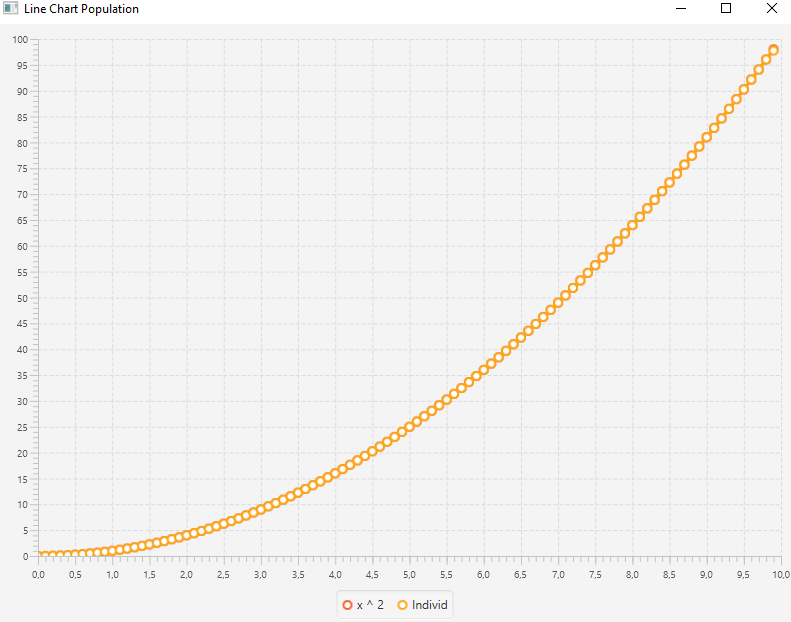


Рисунок – Графики функций, искомой (синий) и полученной (жёлтый)

1. Листинг реализации генетического алгоритма
2. import javafx.application.Application;  
   import javafx.scene.Scene;  
   import javafx.scene.chart.LineChart;  
   import javafx.scene.chart.NumberAxis;  
   import javafx.scene.chart.XYChart;  
   import javafx.stage.Stage;  
     
   import java.text.DecimalFormat;  
   import java.util.ArrayList;  
   import java.util.\*;  
     
   public class Main extends Application {  
     
    private static final int *countSigns* = 100;  
    private static final int *countPop* = 20;  
     
    private static ArrayList<Individual> *populations*;  
     
    public static void main(String[] args) {  
    *launch*(args);  
    }  
     
    @Override  
    public void start(Stage primaryStage)  
    {  
    *createPopulation*(); //создание популяции  
     
    Individual min = *getMinIndividual*();  
    System.*out*.println(min.distance);  
     
    int step = 0;  
    while (min.distance >= 0.5)  
    {  
    *reproduction*(); //скрешивание  
    *extinction*(); //вымирание  
     
    min = *getMinIndividual*();  
    System.*out*.println(step+" "+min.distance);  
    step++;  
    }  
     
    showChart(primaryStage, min);  
    }  
     
    public static void createPopulation()  
    {  
    *populations* = new ArrayList<Individual>();  
    for(int i = 0; i < *countPop*; i++)  
    {  
    Double[] signs = new Double[*countSigns*];  
    for(int k = 0; k < *countSigns*; k++)  
    {  
    DecimalFormat df = new DecimalFormat("#.##");  
    double start = new Random().nextInt(10)\*0.1;  
     
    signs[k] = start;  
    }  
     
    Individual individ = new Individual(signs);  
    individ.calculateDistances();  
     
    *populations*.add(individ);  
    }  
    }  
     
    public static Individual getMinIndividual()  
    {  
    return Collections.*min*(*populations*, Comparator.*comparing*(s->s.distance));  
    }  
     
    public static void reproduction()  
    {  
    ArrayList<Individual> parentOne = new ArrayList<Individual>();  
    ArrayList<Individual> parentTwo = new ArrayList<Individual>();  
    ArrayList<Individual> children = new ArrayList<Individual>();  
     
    for(int i = 0 ; i< *populations*.size(); i++)  
    {  
    Individual parent = *populations*.get(i);  
    if(i % 2 == 0)  
    parentOne.add(parent);  
    else  
    parentTwo.add(parent);  
    }  
     
    for(int i = 0 ; i< parentOne.size(); i++)  
    {  
    ArrayList<Individual> child = *Crossbreeding*(parentOne.get(i), parentTwo.get(i));  
    children.addAll(child);  
    }  
     
    ArrayList<Individual> childrenMutate = *mutate*(children);  
     
    for(Individual child:childrenMutate)  
    child.calculateDistances();  
     
    *populations*.addAll(childrenMutate);  
    }  
     
    private static ArrayList<Individual> Crossbreeding(Individual parentOne, Individual parentTwo)  
    {  
    Double[] parentOneSings = parentOne.signs;  
    Double[] parentTwoSings = parentTwo.signs;  
     
    Double[] childOneSings = new Double[*countSigns*];  
    Double[] childTwoSings = new Double[*countSigns*];  
     
    for(int i = 0; i < *countSigns*; i++)  
    {  
    String binaryParentOne = Long.*toBinaryString*(Double.*doubleToRawLongBits*(parentOneSings[i]));  
    String binaryParentTwo = Long.*toBinaryString*(Double.*doubleToRawLongBits*(parentTwoSings[i]));  
     
    String binaryChildOne = binaryParentOne.substring(0, binaryParentOne.length()/2) +  
    binaryParentTwo.substring(binaryParentTwo.length()/2, binaryParentTwo.length());  
     
    String binaryChildTwo = binaryParentTwo.substring(0, binaryParentTwo.length() / 2) +  
    binaryParentOne.substring(binaryParentOne.length()/2, binaryParentOne.length());  
     
    childOneSings[i] = Double.*longBitsToDouble*(Long.*parseLong*(binaryChildOne, 2));  
    childTwoSings[i] = Double.*longBitsToDouble*(Long.*parseLong*(binaryChildTwo, 2));  
    }  
     
    Individual childOne = new Individual(childOneSings);  
    Individual childTwo = new Individual(childTwoSings);  
     
    ArrayList<Individual> childs = new ArrayList<Individual>();  
    childs.add(childOne);  
    childs.add(childTwo);  
     
    return childs;  
    }  
     
    public static ArrayList<Individual> mutate(ArrayList<Individual> childrens)  
    {  
    ArrayList<Individual> mutateChildrens = new ArrayList<Individual>(childrens);  
    for(Individual mutate:mutateChildrens)  
    {  
    for(int i = 0; i < *countSigns*; i++)  
    {  
    Double x = i \* 0.1;  
    Double y = MathAlgorithm.*Function*(x);  
     
    if(mutate.signs[i] > y && (mutate.signs[i] - y > 0.005))  
    {  
    mutate.signs[i] -= 0.005;  
    }  
     
    if(mutate.signs[i] < y && (y - mutate.signs[i] > 0.005))  
    mutate.signs[i] += 0.005;  
    }  
    }  
     
    return mutateChildrens;  
    }  
     
    private static void extinction()  
    {  
    int count = *populations*.size()/2;  
    ArrayList<Individual> survivors = new ArrayList<Individual>();  
    for(int i = 0; i < count; i++)  
    {  
    Individual survivor = *getMinIndividual*();  
    survivors.add(survivor);  
    *populations*.remove(survivor);  
    }  
    *populations*.clear();  
    *populations*.addAll(survivors);  
    }  
     
    private void showChart(Stage stage, Individual min)  
    {  
    Double[] functionY = new Double[*countSigns*];  
    Double[] populationY = new Double[*countSigns*];  
    for(int i = 0; i < *countSigns*; i++)  
    {  
    Double x = i \* 0.1;  
    functionY[i] = MathAlgorithm.*Function*(x);  
    populationY[i] = min.signs[i];  
     
    System.*out*.println(populationY[i]);  
    }  
     
     
    stage.setTitle("Line Chart Population");  
     
    final NumberAxis xAxis = new NumberAxis();  
    final NumberAxis yAxis = new NumberAxis();  
     
    final LineChart<Number,Number> lineChart =  
    new LineChart<Number,Number>(xAxis,yAxis);  
     
     
    XYChart.Series funSeries = new XYChart.Series();  
    XYChart.Series popSeries = new XYChart.Series();  
     
    for(int i = 0; i < *countSigns*; i++) {  
    Double x = i \* 0.1;  
    funSeries.getData().add(new XYChart.Data(x, functionY[i]));  
    popSeries.getData().add(new XYChart.Data(x, populationY[i]));  
    }  
    funSeries.setName("x ^ 2");  
    popSeries.setName("Individ");  
     
     
    Scene scene = new Scene(lineChart,800,600);  
    lineChart.getData().add(funSeries);  
    lineChart.getData().add(popSeries);  
     
    stage.setScene(scene);  
    stage.show();  
     
    }  
   }

ЗАКЛЮЧЕНИЕ

В результате выполнения работы был реализован генетический алгоритм, а именно: алгоритм размножения особей, алгоритм мутации, алгоритм отбора лучших особей. получены навыки работы со средствами реализации подобного алгоритма на языке Java.