МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
МЕХАНИКО-МАТЕМАТИЧЕСКИЙ ФАКУЛЬТЕТ
КАФЕДРА ВЕБ-ТЕХНОЛОГИЙ И КОМПЬЮТЕРНОГО МОДЕЛИРОВАНИЯ

ЭВОЛЮЦИОННАЯ ТЕОРИЯ И АЛГОРИТМЫ

Курсовая работа

Жаврид Марии Сергеевны
студентки 2 курса,
специальность 1-31 03 08-01
Математика и информационные технологии (веб-программирование и интернет-технологии)
Научный руководитель:
Ассистент
В. Р. Харитонова

Минск, 2020

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В прошлом на мировоззрение людей сильное влияние оказала теория эволюции Чарльза Дарвина, впервые представленная в работе “Происхождение видов”, в 1859 году. В своих трудах Дарвин описывал механизмы развития, главным из которых является отбор в соединении с изменчивостью.

Идея эволюционных алгоритмов и, как частный случай, генетических алгоритмов заимствована у живой природы и состоит в организации эволюционного процесса, конечной целью которого является получения оптимального решения в сложной комбинаторной задаче. Впервые эти нестандартные идеи были применены к решению оптимизационных задач в середине семидесятых. После выхода книги, ставшей классикой - «Адаптация в естественных и искусственных системах» ("Adaptation in Natural and Artifical Systems", 1975), направление получило общее признание.

Область применения генетических алгоритмов достаточно обширна. Они успешно используются для решения больших и экономически значимых задач в бизнесе и инженерных разработках. С помощью генетических алгоритмов были разработаны промышленные проектные решения, позволившие сэкономить многомиллионные суммы. Они также используются для обучения нейронных сетей и получения исходных данных для работы других алгоритмов поиска и оптимизации. Именно поэтому проблематика исследования по теме «Генетические алгоритмы» несет актуальный характер.

Объектом исследования данной курсовой работы являются генетические алгоритмы.

Целью работы является изучение основных аспектов генетических алгоритмов и область их применения.

Методы исследования:

- сбор и анализ литературных источников по данной теме;
- моделирование работы генетического алгоритма на компьютере и его реализация.

Задачи:

1. Изучить принципы работы, основных этапов и параметров генетических алгоритмов.
2. Реализовать генетический алгоритм для решения задачи целочисленного программирования.
3. Проанализировать полученные результаты.

ГЛАВА 1. ГЕНЕТИЧЕСКИЕ АЛГОРИТМЫ

1.1. Перенос теории эволюции на алгоритмы

Генетические алгоритмы — это класс эволюционных алгоритмов поиска. Они являются частью группы методов, называемой эволюционными вычислениями, которые объединяют различные варианты использования эволюционных принципов для достижения поставленной цели.

Идея генетических алгоритмов основана на эволюционной теории Чарльза Дарвина, представленной в работе “Происхождение видов”, в 1859 году. Эти алгоритмы симулируют процесс естественного отбора, когда более сильные особи из популяции переживают более слабых и производят следующее поколение особей. При этом сохраняется биологическая терминология в упрощенном виде и основные понятия линейной алгебры.

Главная трудность при построении вычислительных систем, основанных на принципах естественного отбора и применении этих систем в прикладных задачах, состоит в том, что естественные системы довольно хаотичные, а все наши действия, фактически, носят четкую направленность. Выживание в природе не направлено к фиксированной цели, вместо этого эволюция делает шаг вперед в любом доступном направлении. Системы, направленные на моделирование эволюции по аналогии с естественными системами можно разбить на две большие категории:

1. Системы, смоделированные на биологических принципах. Они успешно используются для задач функциональной оптимизации и могут легко быть описаны небиологическим языком.
2. Системы, которые биологически более правдоподобны, но на практике неэффективны.

К первой категории мы и относим эволюционные алгоритмы, генетические алгоритмы и эволюционные стратегии.

Генетические алгоритмы являются разновидностью методов поиска с элементами случайности. Их цель заключается в нахождении лучшего решения по сравнению с имеющимся, а не оптимального решения задачи. Это связано с тем, что для сложной системы часто требуется найти хоть какое-нибудь удовлетворительное решение, а проблема достижения оптимума отходит на второй план. При этом другие методы, ориентированные на поиск именно оптимального решения, вследствие чрезвычайной сложности задачи становятся вообще неприменимыми. В этом кроется причина появления, развития и роста

популярности генетических алгоритмов. Хотя, как и всякий другой метод поиска, этот подход не является оптимальным методом решения любых задач. Дополнительным свойством этих алгоритмов является невмешательство человека в развивающийся процесс поиска. Человек может влиять на него лишь опосредованно, задавая определенные параметры.

В генетическом алгоритме предварительно анализируется множество входных параметров оптимизационной задачи и находится некоторое множество альтернативных решений. Каждое решение кодируется как последовательность конечной длины. Во всех моделях эволюции используется принцип выживания сильнейших, т.е. наименее подходящие решения устраняются, а лучшие переходят в следующую генерацию. Алгоритм работает пока не будет получено решение заданного качества или не будет выполнено заданное количество генераций. Он также может остановится если на некоторой генерации возникает преждевременная сходимость, когда найден локальный оптимум.

Генетические алгоритмы применяются для решения следующих задач:

- оптимизация функций;
- разнообразные задачи на графах (задача коммивояжера, раскраска);
- настройка и обучение искусственной нейронной сети;
- задачи компоновки;
- составление расписаний;
- игровые стратегии;
- аппроксимация функций;
- биоинформатика.

В данной работе будет рассматриваться реализация генетического алгоритма для задачи целочисленного программирования (поиск минимума функции на некоторой области допустимых значений).

1.2. Основные этапы генетического алгоритма. Принцип действия

Прежде чем рассматривать непосредственно работу генетического алгоритма, нужно выделить некоторый термины, широко используемые в данной области.

Хромосома - вектор (или строка), состоящий из каких-либо чисел. Если хромосома представлена бинарной строкой из нулей и единиц, то каждая ее позиция называется *геном*.

Индивидуум (особь) - набор хромосом (вариант решения задачи). Обычно особь состоит из одной хромосомы, поэтому индивидуум и хромосома в дальнейшем будут использоваться как идентичные понятия.

Кроссинговер (кроссовер) - операция, при которой две хромосомы обмениваются своими частями.

Мутация - случайное изменение одной или нескольких позиций в хромосоме.

Популяция - совокупность индивидуумов.

Пригодность (приспособленность) - критерий или функция, экстремум которой следует найти.

Хеммингово расстояние - число различающихся в обоих хромосомах компонент (ген).

Генетический алгоритм состоит из нескольких *этапов*:

1. Инициализация.
2. Оценивание.
3. Выбор родителей.
4. Кроссинговер.
5. Мутация.
6. Обновление популяции.

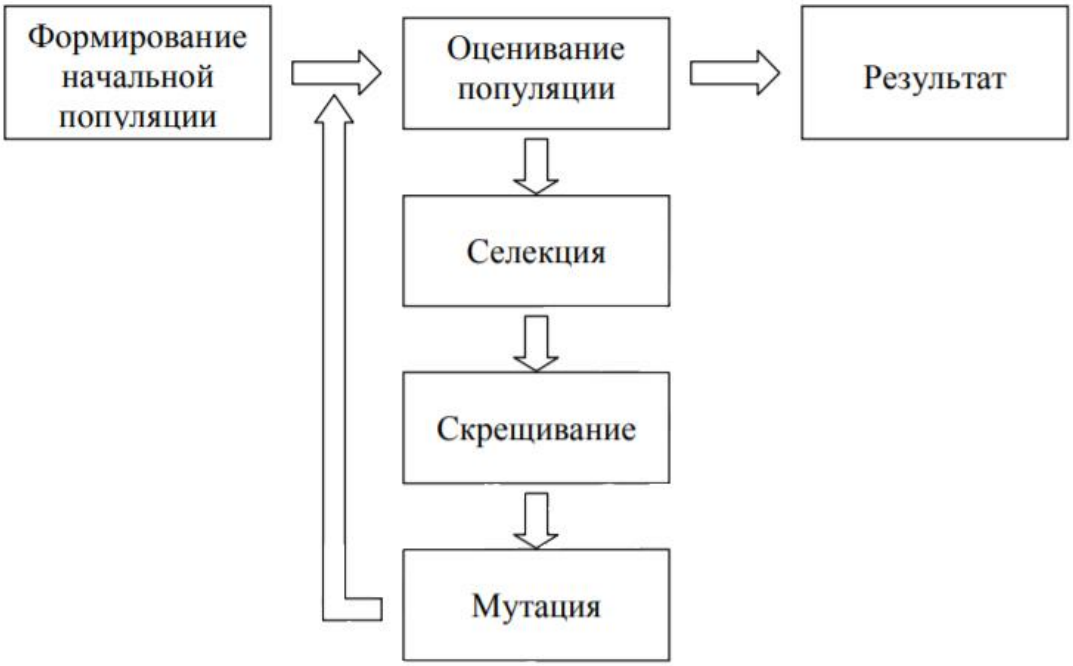

Рисунок 1: Схема генетического алгоритма

Рассмотрим подробнее каждый из этапов.

На первом шаге происходит генерация начальной популяции. Выбирается определенное количество пробных решений, записанных в двоичной форме, которые в последствии и называются хромосомами.

Затем выполняется оценивание, т.е. вычисление пригодности каждой хромосомы. Как известно, принцип естественного отбора заключается в том, что в конкурентной борьбе выживает наиболее приспособленный. В задаче

целочисленного программирования приспособленность особи определяется целевой функцией.

Во время третьего этапа из текущей популяции отбирается *брачная пара* индивидов, которые будут в дальнейшем скрещиваться (важно заметить, что выбор особей-родителей выполняется случайным образом). Процесс кроссинговера заключается в выборе точек разрыва альтернативных решений и обмене участками хромосом между родителями.

После скрещивания проводится мутация потомков: выполнение (обычно небольших) изменений в значениях одного или нескольких генов в хромосоме. В генетических алгоритмах мутация рассматривается как метод восстановления потерянного генетического материала, а не как поиск лучшего решения. Вероятность мутации может являться или фиксированным случайным числом на отрезке [0; 1], или функцией от какой-либо характеристики решаемой задачи. Для особей, кодированных двоичным кодом, мутация заключается в случайном инвертировании гена (0 заменяется 1 и наоборот).

В конце происходит обновление популяции: создается новая популяция с помощью различных методов отбора особей. Затем шаги повторяются.

Количество запусков алгоритма выбирается в зависимости от условия задачи или в зависимости от критерия остановки алгоритма. Критерий может быть различным: нахождение глобального решения, исчерпание числа поколений, отпущенных на эволюцию, или исчерпание времени, отпущенного на эволюцию.

Таким образом, через несколько прогонов алгоритма мы получим поколение из похожих и наиболее приспособленных особей. Значение приспособленности наиболее “хорошей” особи и будет являться решением задачи.

На различных этапах генетического алгоритма методы и подходы могут варьироваться в зависимости от условия задачи. В дальнейшем будут рассматриваться модифицированные операторы и модели самого алгоритма.

1.2.1. Операторы выбора родителей

Существует несколько подходов к выбору родительской пары особей.

Наиболее распространёнными являются следующие.

Панмиксия - наиболее простой оператор выбора. В соответствии с ним каждому члену популяции сопоставляется случайное целое число на отрезке [1; n], где n - количество особей в популяции. Эти числа в дальнейшем рассматриваются как номера особей, которые примут участие в скрещивании. При таком выборе какие-то из членов популяции не будут участвовать в процессе размножения, так как образуют пару сами с собой. Какие-то члены

популяции примут участие в процессе воспроизводства неоднократно. Несмотря на простоту, такой подход универсален для решения различных классов задач. Однако стоит заметить, что с ростом численности популяции эффективность алгоритма, реализующего такой подход, снижается.

Инбридинг представляет собой такой метод, при котором первый родитель выбирается случайным образом, а вторым родителем становится член популяции, ближайший к первому. Здесь “ближайший” может пониматься, например, в смысле минимального расстояния Хемминга.

При *аутбридинге* также используется понятие схожести особей. Однако теперь пары формируются из максимально далеких особей.

Инбридинг и аутбридинг бывают *генотипными*, если в качестве расстояния берется разность значений целевой функции для соответствующих особей, и *фенотипным*, если в качестве расстояния берется расстояние Хемминга.

Селекция состоит в том, что родителями могут стать только те особи, значение приспособленности которых не меньше пороговой величины. Такой подход обеспечивает более быструю сходимость алгоритма, однако он не подходит тогда, когда, к примеру, ставится задача определения нескольких экстремумов. Пороговая величина в селекции может быть вычислена разными способами, наиболее известные из которых — это *турнирный* и *рулеточный* отборы.

1.2.2. Кроссинговер

При реализации генетического алгоритма одним из важнейших вопросов является определение места и количества точек разрыва хромосомы альтернативных решений. При этом большое количество точек разрыва может привести к полной потере лучших решений, а маленькое часто приводит к попаданию решения в локальный оптимум, далекий от глобального. Поэтому в зависимости от поставленной задачи необходимо правильно выбрать вид кроссинговера.

При *одноточечном* кроссинговере в родительских особях случайным образом определяется точка разрыва, в которой обе хромосомы делятся на две части и обмениваются ими. Такой тип скрещивания называется одноточечным, так как при нем родительские хромосомы разделяются только в одной случайной точке.

В *двухточечном* кроссинговере (рис.1) хромосомы рассматриваются как циклы, которые формируются соединением концов линейной хромосомы вместе. Для замены сегмента одного цикла сегментом другого цикла требуется выбор двух точке разрыва. Следовательно, двухточечный кроссинговер решает

ту же самую задачу, что и одноточечный, но более полно. В настоящий момент многие исследователи соглашаются, что двухточечный кроссинговер вообще лучше, чем одноточечный.

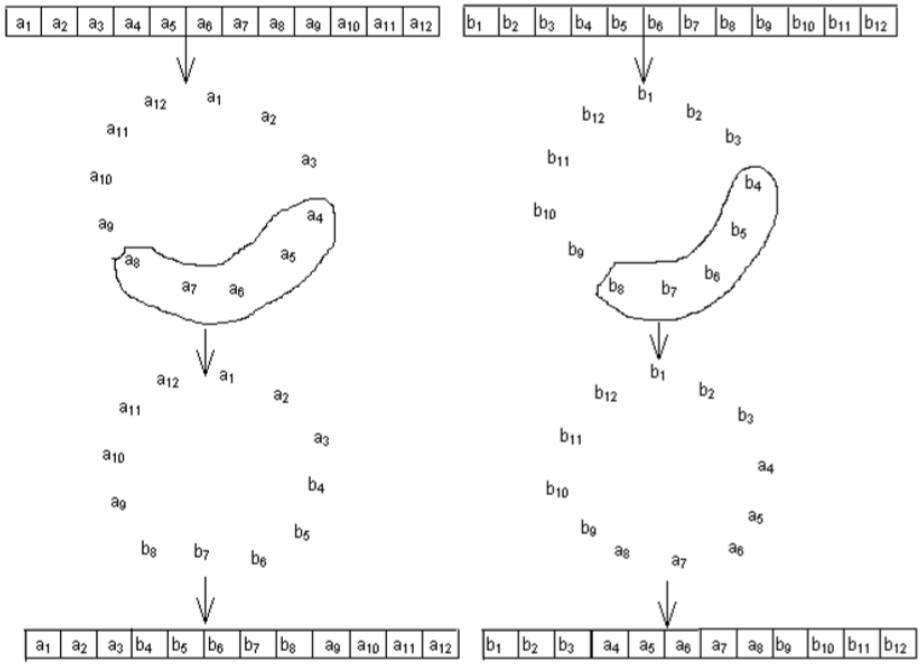

Рисунок 2: Двухточечный кроссинговер

Для *многоточечного* кроссинговера случайно без повторений выбирается m точек разреза. Затем они сортируются в порядке возрастания. При таком кроссинговере происходит обмен участками хромосом, ограниченными точками разреза и таким образом получаются два потомка. Применение многоточечного кроссинговера требует введения нескольких переменных для точек разреза, и для воспроизведения выбираются особи с наибольшей приспособленностью.

При *перетасовочном* кроссинговере особи, отобранные для скрещивания, случайным образом обмениваются генами, а затем выбирается точка для одноточечного кроссинговера и проводится обмен частями хромосом. После скрещивания потомки вновь тусуются. Таким образом, при каждом кроссинговере создаются не только новые потомки, но и модифицируются родители, что позволяет сократить число операций по сравнению с однородным кроссинговером.

1.2.3. Операторы отбора особей в новую популяцию

Для создания новой популяции можно использовать различные методы отбора особей. Этап отбора хромосом в новую популяцию является одним из важнейших, так как он непосредственно влияет на скорость работы алгоритма.

Наиболее эффективные методы — это отбор *вытеснением* и *элитарный* отбор.

При *элитарном отборе* создается промежуточная популяция, которая включает в себя как родителей, так и их потомков. Члены этой популяции оцениваются, а затем из них выбираются N самых лучших (пригодных), которые и войдут в следующее поколение. Использование данного механизма отбора оказывается весьма полезным для эффективности генетического алгоритма, так как не допускает потерю лучших решений: полученное хорошее решение будет оставаться в популяции до тех пор, пока не будет найдено еще лучшее. Однако при элитарном отборе может возникнуть потенциальная опасность преждевременной сходимости. Когда это необходимо, быстрая сходимость может быть с успехом компенсирована подходящим оператором выбора родительский пар. К примеру, комбинация “аутбридинг - элитарный отбор” является одной из наиболее эффективных.

В *отборе вытеснением* выбор особи в новую популяцию зависит не только от величины ее приспособленности, но и от того, есть ли уже в формируемой популяции особь с аналогичным хромосомным набором. Отбор проводится из числа родителей и их потомков. Из всех особей с одинаковой приспособленностью предпочтение сначала отдается особям с разными генотипами. Таким образом, возникают два преимущества: во-первых, лучшие найденные решения, обладающие различными хромосомными наборами, не теряются, а во-вторых, в популяции постоянно присутствует генетическое разнообразие. Отбор вытеснением в данном случае формирует популяцию скорее из далеко расположенных особей, вместо особей, группирующихся около текущего найденного решения. При использовании этого метода помимо определения глобальных минимумов появляется возможность выделить и те локальные минимумы, значения которых близки к глобальным.

1.3. Некоторые модели генетических алгоритмов

Важно заметить, что в настоящее время генетические алгоритмы — это целый класс алгоритмов, созданных для решения разнообразных задач.

1.3.1. Канонический ГА

Рассмотрим модель генетического алгоритма, называемую *канонической*.

Данная модель является классической. Она была предложена Джоном Холландом в его знаменитой работе “Адаптация в природных и искусственных средах” (1975).

Канонический генетический алгоритм имеет следующие характеристики:

- целочисленное кодирование;

• все хромосомы в популяции имеют одинаковую длину;
• постоянный размер популяции;
• пропорциональный отбор;
• новое поколение формируется только из особей-потомков;
• одноточечный кроссовер;
• битовая мутация.

1.3.2. Генитор

В модели генитор (Genitor) используется специфичная стратегия отбора. Вначале, как и полагается, популяция инициализируется, и ее особи оцениваются. Затем выбираются случайным образом две хромосомы и скрещиваются. В итоге получается только один потомок, который оценивается и занимает место менее приспособленной особи в популяции. После этого снова случайным образом выбираются две родительской особи, и их потомок занимает место особи с самой низкой приспособленностью. Таким образом, на каждом шаге в популяции обновляется только одна особь. Процесс продолжается до тех пор, пока пригодности хромосом не станут одинаковыми. В данный алгоритм можно добавить мутацию потомка после его создания. Критерий окончания процесса, как и вид кроссинговера и мутации, можно выбирать разными способами.

Можно выделить следующие характеристики данного алгоритма:

- постоянный размер популяции;
- все хромосомы в популяции имеют одинаковую длину;
- особи для скрещивания выбираются случайным образом;
- результатом скрещивания является только один потомок;
- мутация может отсутствовать;

1.3.3. Гибридный алгоритм

Использование гибридного алгоритма позволяет объединить преимущества генетических алгоритмов и преимущества некоторых других классических методов, подходящих для выбранной задачи. Известно, что генетический алгоритм может испытывать трудности в получение наилучших решений из всех “хороших”. Сочетание двух алгоритмов позволяет избежать проблем с поиском глобального оптимума и делает возможным использование преимуществ обоих.

В каждом поколении все сгенерированные потомки оптимизируются выбранным вспомогательным методом и затем заносятся в новую популяцию. Тем самым получается, что каждая особь в популяции достигает локального

оптимума, вблизи которого она находится. Далее реализуются обычные этапы генетического алгоритма.

Характеристики алгоритма:

- все хромосомы в популяции имеют одинаковую длину;
- постоянный размер популяции;
- могут использоваться любые комбинации стратегий отбора и формирования следующего поколения;
- ограничений на типа кроссовера или мутации нет.

1.4. Преимущества и недостатки генетических алгоритмов

Основными преимуществами генетических алгоритмов является концептуальная простота и простота реализации. Сами алгоритмы состоят из пяти довольно простых шагов: инициализация, оценка пригодности особей, селекция, скрещивание и мутация. Следует также отметить, что генетическим алгоритмам не требуется никакой информации о поведении функции, и они могут быть использованы в задачах с изменяющейся средой.

Естественно, имеются и недостатки. Генетические алгоритмы довольно трудно смоделировать для нахождения всех решений задачи, и всегда есть вероятность попасть в локальный оптимум, а не найти точный глобальный. Далеко не все задачи можно адаптировать под генетический алгоритм и найти оптимальное кодирование параметров. Еще одним недостатком является трудность с определением критерия окончания работы алгоритма: зачастую невозможно реализовать его эффективно. Также необходимо учитывать различные аспекты, зависящие от условия задачи и влияющие на скорость работы алгоритма.

ГЛАВА 2. ПРАКТИЧЕСКОЕ ПРИМЕНЕНИЕ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ. РЕШЕНИЕ ЗАДАЧИ ЦЕЛОЧИСЛЕННОГО ПРОГРАММИРОВАНИЯ С ПОМОЩЬЮ ГЕНЕТИЧЕСКИХ АЛГОРИТМОВ

2.1. Описание программы

2.1.1. Входные данные

В ходе курсовой работы я реализовала два генетических алгоритма: классический ГА и генитор. Алгоритмы были написаны на языке Java.

Задача заключалась в следующем: найти минимум функции одной переменной на некоторой области допустимых решений при условии, что все переменные функции принимают только целые значения и имеют ограничения.

Входные данные:

Хромосомы будут записываться в виде массива, состоящего из нулей и единиц (бинарный код).

`CROSSOVER_RATE = 0.05;` - постоянная, которая будет использоваться при кроссинговере.

`CHROMOSOME_LENGTH = 3;` - длина одной хромосомы (особи).

`GENE_LENGTH=3;` - количество позиций в хромосоме (в сущности, то же самое что и длина хромосомы).

`MUTATION_RATE = 0.15;` - постоянная, которая будет использоваться при мутации; отвечает за вероятность мутации одной или нескольких позиций в хромосоме.

`TOURNAMENT_SIZE = 5;` - размер популяции.

`SIMULATION_LENGTH=100;` - количество повторений цикла.

[0, 7] - отрезок, ограничивающий область допустимых значений. Так как длина хромосомы 3, максимальным числом в двоичной системе счисления будет 111, т.е. 7.

2.1.2. Основные классы

В пакете entity находятся следующие основные классы:

1. Constant
2. Chromosome
3. Population

В классе Constant находятся основные константные входные значения, представленные выше.

```java
public class Constant {
    private Constant(){}
    public static final double CROSSOVER_RATE = 0.05;
    public static final double MUTATION_RATE = 0.15;
    public static final int TOURNAMENT_SIZE = 5;
    public static final int CHROMOSOME_LENGTH = 3;
    public static final int GENE_LENGTH = 3;
    public static final int SIMULATION_LENGTH=100;

}
```

Рисунок 3: Класс Constant

Класс **Chromosome** имеет следующие поля:
`private int[] genes;` - массив, состоящий из нулей и единиц.
`private double fitness;` - оценочное значение одной хромосомы.
`private Random randomGenerator;` - объект класса Random для генерации псевдослучайных чисел.

Основными функциями являются:
`public Chromosome()` - конструктор класса, выделяющий место в памяти под массив генов и создающий объект randomGenerator.
`public void generateIndividual()` - генерирует массив генов, заполняя его единицами и нулями.
`public double f(int x)` - возвращает значение, которое приобретает функция при заданном значении x.
`public int genesToInt()` - данная функция переводит величину хромосомы из двоичной системы счисления в десятичную.
`public double getFitness()` - функция оценивания, которая возвращает значение заданной функции в определенной точке.

Следующий класс (**Population**) определяет популяцию и имеет всего одно поле:
`private Chromosome[] chromosomes;` - массив из объектов класса Chromosome.

Основными функциями здесь являются `initialize()` и `getFittestIndividual()`.

```java
public void initialize(){
 for(int i = 0; i< chromosomes.length; ++i){
  Chromosome newChromosome = new Chromosome();
```

```
  newChromosome.generateIndividual();
  saveIndividual(i, newChromosome);
 }
}
```

В указанной выше функции происходит инициализация популяции. Сначала создается объект класса Chromosome, затем с помощью функции, генерирующей особь, массив генов этого объекта заполняется нулями и единицами случайным образом. Функция saveIndividual сохраняет получившуюся хромосому в популяцию.

```
public Chromosome getFittestIndividual(){
Chromosome fittest = chromosomes[0];
 for(int i = 1; i< chromosomes.length; ++i){
  if(getIndividual(i).getFitness() < fittest.getFitness())
  fittest = getIndividual(i);
 }
 return fittest;
}
```

Данная функция предназначена для определение лучшего решения среди всех решений в популяции. Оценочные значения каждой хромосомы сравниваются друг с другом, а затем объект с наименьшим из них и возвращает функция.

2.2. Классы генетических алгоритмов

Как говорилось ранее, в ходе выполнения данной работы я реализовала два алгоритма. Для каждого из них был разработан отдельный класс.

Первый алгоритм - *классический* - представлен в классе ClassicGeneticAlgorithm.java, второй - *genitor* - представлен в GenitorGeneticAlgorithm.java.

2.2.1. ClassicGeneticAlgorithm

Главной функцией является evolvePopulation, в которой генерируется новая популяция с помощью остальных функций класса. Сначала выбирается брачная пара с помощью функции отбора. Затем функция кроссинговера возвращает нового потомка. Все получившиеся решения заносятся в новую популяцию, состоящую только из особей-потомков. После происходит мутация и отбор наилучших решений в новое поколение из родительской популяции и популяции потомков.

Рассмотрим основные функции подробнее.

1. Функция, выбирающая родительскую особь в случайном порядке:

```
public Chromosome randomSelection(Population population){
 int  randomIndex = (int)
(Math.random()*population.size());
 return population.getIndividual(randomIndex);
}
```

Здесь генерируется произвольное число в диапазоне от 0 до 5 и возвращается особь популяции с получившемся индексом.

2. Функция кроссинговера:

```
public Chromosome crossover(Chromosome firstChromosome,
Chromosome secondChromosome){
    Chromosome newSolution = new Chromosome();
 for(int i=0; i< Constant.CHROMOSOME_LENGTH; ++i){
 if(Math.random() <= Constant.CROSSOVER_RATE){
 newSolution.setGene(i, firstChromosome.getGene(i));
 }  else{
 newSolution.setGene(i, secondChromosome.getGene(i));
 }
    }
    return newSolution;
}
```

В данном случае использовался унифицированный кроссинговер: в случайном порядке определяется чей родительский ген займет следующую позицию в хромосоме. Функция возвращает новую особь-потомка.

3. Мутация потомка:

```
public void mutate(Chromosome chromosome){
for(int i = 0; i< Constant.CHROMOSOME_LENGTH; ++i){
 if(Math.random() <= Constant.MUTATION_RATE){
  int gene = randomGenerator.nextInt(2);
  chromosome.setGene(i,gene);
  }
 }
}
```

Здесь в произвольном порядке определяется одна или более позиций в хромосоме, которая в дальнейшем будет мутировать. Значение гена может изменится, либо остаться таким же.

4. Функция отбора в новую популяцию:

```
public Population selection(Population population, Population
```

```
children){
 Population newPopulation =new
Population(Constant.TOURNAMENT_SIZE*2);
 for(int i=0; i< population.size();i++){
 newPopulation.saveIndividual(i,population.getIndividual(i));
 }
 for(int i=population.size(); i< newPopulation.size();i++){
 newPopulation.saveIndividual(i,children.getIndividual(i-
Constant.TOURNAMENT_SIZE));
 }
 sort(newPopulation);
Population sortedPopulation = new
Population(population.size());
 for(int i=0; i< population.size();i++){

sortedPopulation.saveIndividual(i,newPopulation.getIndividual(
i));
}
    return sortedPopulation;
}
```

Данная функция сначала создает сборное поколение, состоящее из потомков и родительских особей. Далее функция *sort(newPopulation)* сортирует сборную популяцию по оценочному признаку в порядке убывания. Затем создается и возвращается новая популяция фиксированного размера, в которую заносятся первые n = population.size() особей с лучшими значениями fitness-функции.

2.2.2. GenitorGeneticAlgorithm

Алгоритм генитор отличается от классического тем, что особь-потомок занимает место наислабейшей по оценке родительской особи в исходной популяции.

Основная функция:

```
public Population evolvePopulation(Population population){
 for(int i = 0; i<population.size(); ++i){
 Chromosome firstChromosome = randomSelection(population);
 Chromosome secondChromosome = randomSelection(population);
 Chromosome newChromosome =
crossover(firstChromosome,secondChromosome);
 int index = findWeakest(population);
 population.saveIndividual(index, newChromosome);
```

```
    }
 for(int i=0; i< population.size();++i){
 mutate(population.getIndividual(i));
 }
 return population;
}
```

После кроссинговера выполняется функция *findWeakest* которая возвращает целочисленное значение позиции наислабейшей по оценке особи в популяции:

```
public int findWeakest(Population population){
 int temp=0;
 Chromosome min = population.getIndividual(0);
 for(int i=1; i<population.size();i++){
 if(population.getIndividual(i).getFitness()>min.getFitness())
{
                temp = i;
  min=population.getIndividual(i);
 }
        }
    return temp;
}
```

Осуществляется поиск особи с минимальным значением fitness-функции, а ее индекс заносится в переменную temp.

Еще одним отличие генитора от классического алгоритма является функция выбора родительской особи.

В классическом варианте родитель выбирался в случайном порядке - здесь же сначала создается новая популяция, заполненная хромосомами из родительского поколения, выбранными произвольно (особь-родитель может как входить в новую популяцию несколько раз, так и не входить вообще). Затем из получившейся популяции выбирается особь-родитель с наилучшим оценочным показателем.

```
public Chromosome randomSelection(Population population){
    Population newPopulation = new
Population(Constant.TOURNAMENT_SIZE);
 for(int i=0; i<Constant.TOURNAMENT_SIZE; ++i){
 int  randomIndex = (int) (Math.random()*population.size());

newPopulation.saveIndividual(i,population.getIndividual(random
```

```
Index));
 }
 Chromosome fittestChromosome
=newPopulation.getFittestIndividual();
    return fittestChromosome;
}
```

2.3. Пример работы генетического алгоритма

Пример выполнения классического генетического алгоритма представлен на рис.4. В данном случае использовалась следующая функция:

$f(x)=5-24x+17x^2-\frac{11}{3}x^3+\frac{1}{4}x^4$	(1)

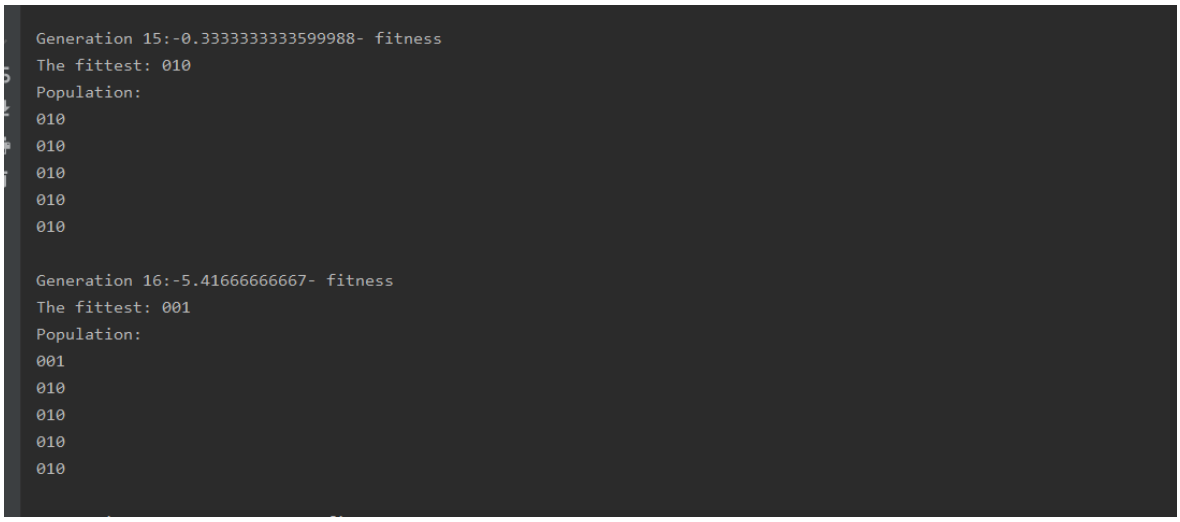

Рисунок 4: Результат выполнения классического ГА

В консоль выводится номер популяции, оценка наилучшей особи, сама наилучшая особь и вся популяция.

График функции (1) представлен на рисунке ниже:

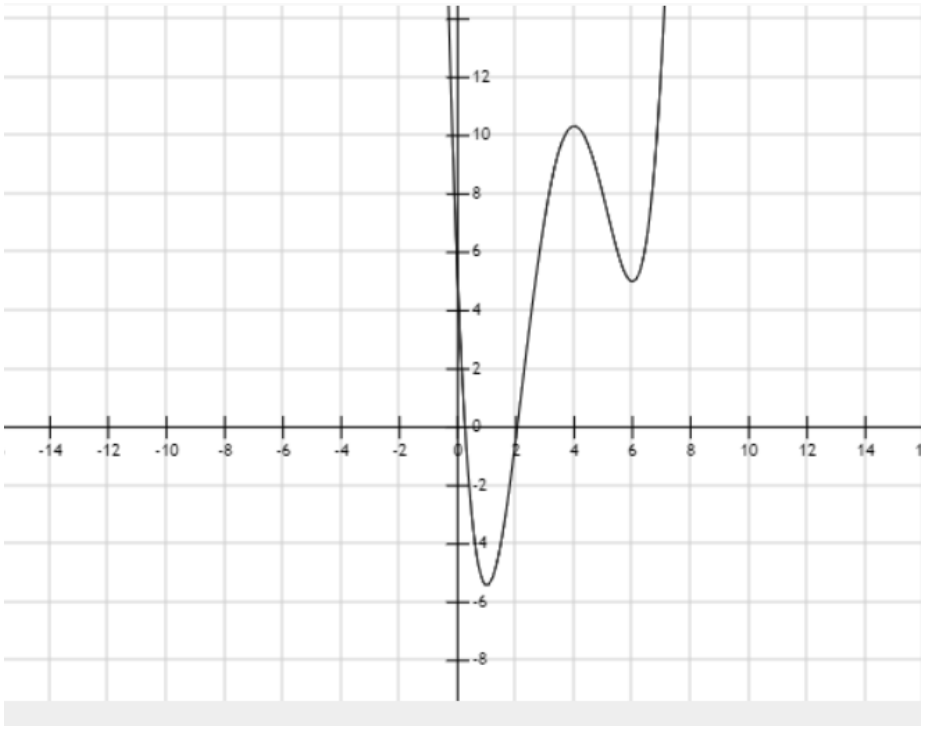

Рисунок 5: График функции (1)

Как можно заметить наилучший результат появляется уже в шестнадцатой популяции и действительно является глобальным минимумом функции (1).

В дальнейшем особи в популяции постепенно будут становится все ближе и ближе к правильному значению, пока в какой-то момент все хромосомы не станут одинаковыми (рис.6).

```
The fittest: 001
Population:
001
001
001
001
001

Generation 79:-5.41666666667- fitness
The fittest: 001
Population:
001
001
001
001
001

Generation 80:-5.41666666667- fitness
The fittest: 001
Population:
001
001
001
001
001

Generation 81:-5.41666666667- fitness
The fittest: 001
```

Рисунок 6: Результат выполнения классического ГА

В отличие от классического алгоритма, генитор справляется гораздо быстрее и показывает нужные результаты уже на третьем шаге, как видно из рис.7.

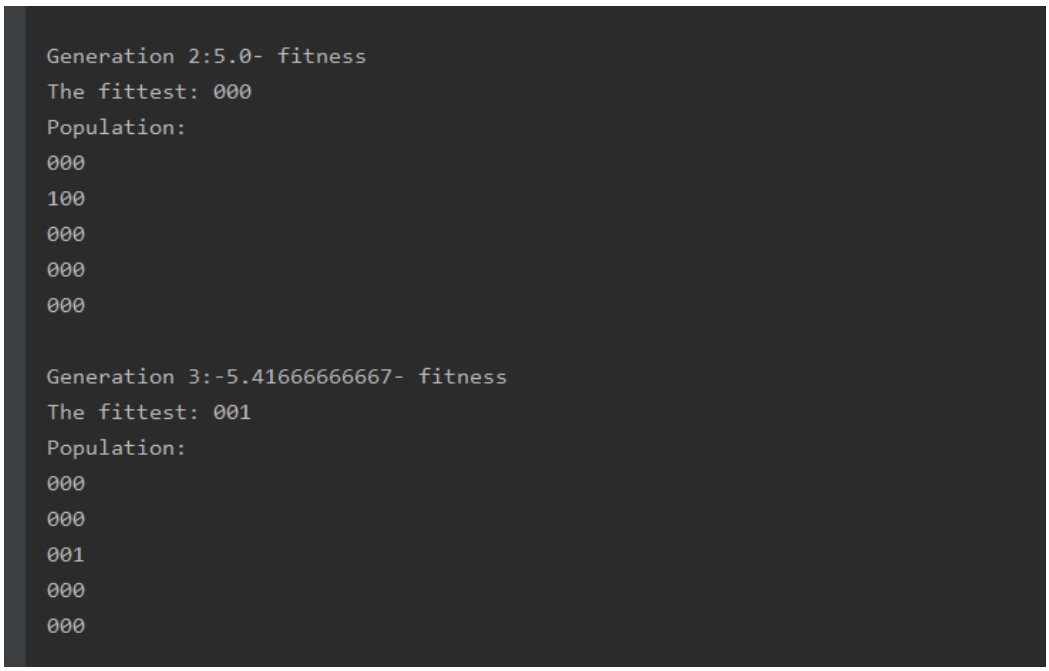

Рисунок 7: Результат выполнения ГА генитор

2.4. Анализ работы генетических алгоритмов для разных типов функций

В данном разделе будут рассматриваться результаты работы двух генетических алгоритмов для следующих функций:

1. Линейная функция.
2. Квадратичная функция.
3. Функция с синусом.

2.4.1. Результаты работы ГА для линейной функции

Здесь целевая функция является простой линейной функцией f(x)=x+2. Ее график выглядит следующим образом:

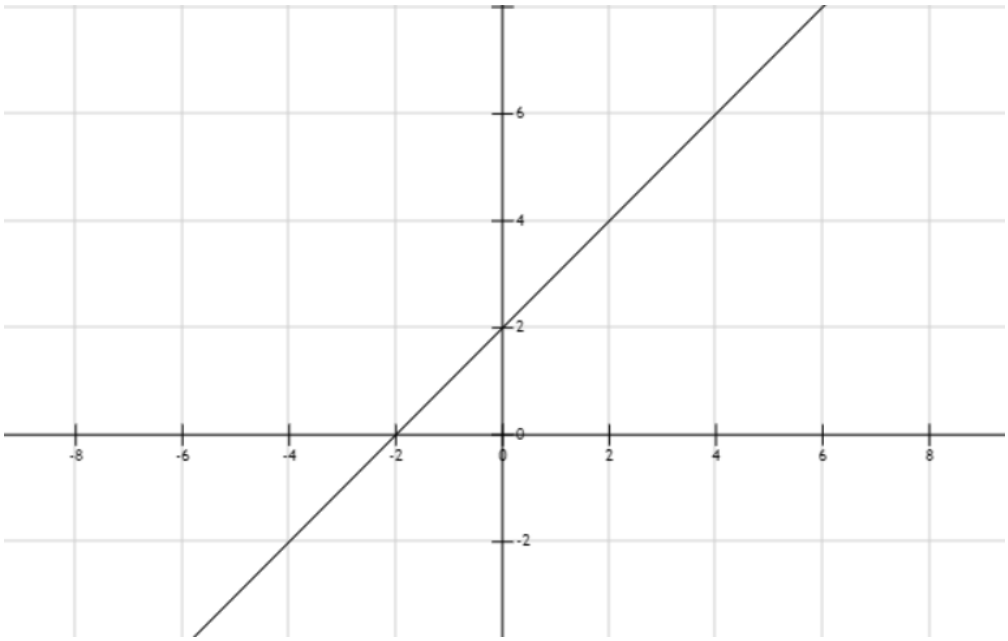

Рисунок 8: График функции f(x)=x+2

Можно отметить, что минимумом данной функции на области допустимых значений [0, 7] является аргумент 2, соответствующий точке 0 (000).

Ниже на рис.9 представлен результат работы классического ГА.

```
Generation 6:3.0- fitness
The fittest: 001
Population:
001
001
001
001
001

Generation 7:2.0- fitness
The fittest: 000
Population:
000
001
001
001
001
```

Рисунок 9: Результаты работы классического ГА для линейной функции

Нужное решение задачи появляется на 7 шаге цикла.

Теперь рассмотрим результат работы генитора для такой же функции.

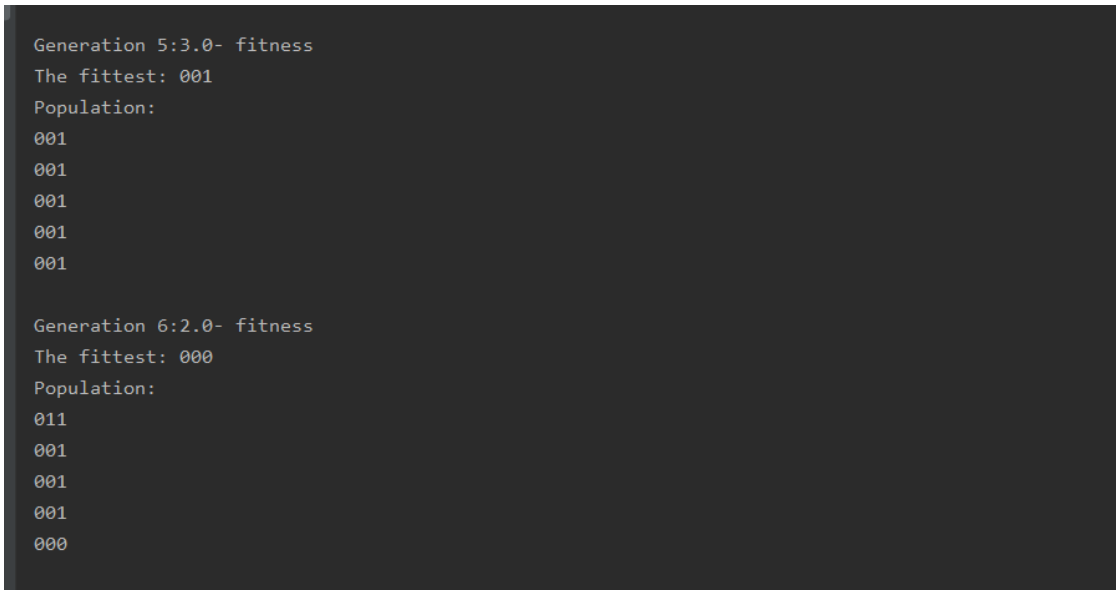

Рисунок 10: Результаты работы ГА генитор для линейной функции

Генитор, как и классический алгоритм, также легко находит нужный минимум, однако искомое решение в данном случае найдено немного быстрее - оно появляется на 6 шаге.

Что классический генетический алгоритм, что генитор одинаково быстро находят решение задачи с линейной целевой функцией без каких-либо затруднений.

2.4.2. Результаты работы ГА для квадратичной функции

В данном случае целевая функция является квадратичной и выглядит следующим образом:

$f(x)=17x^2+3x+1$	(2)

График функции (2) представлен на рис.11.

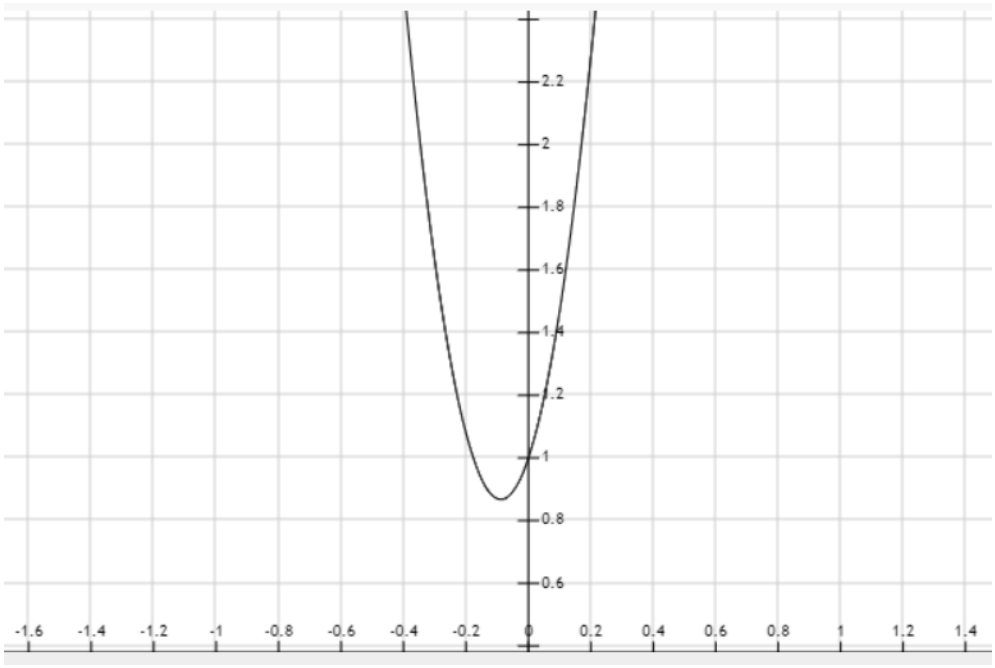

Рисунок 11: График функции (2)

Как можно заметить, глобальный минимум на отрезке [0, 7] находится в точке 1(001).

Теперь проверим работу генетических алгоритмов с целевой квадратичной функцией.

Классический ГА находит ее минимум уже на втором шаге цикла.

```
Generation 1:21.0- fitness
The fittest: 001
Population:
010
010
100
100
001

Generation 2:1.0- fitness
The fittest: 000
Population:
000
001
001
010
010
```

Рисунок 12: Результаты работы классического ГА для функции (2)

Как ни странно, генитор нашел правильное решение гораздо позже - глобальный минимум появляется только в 19 популяции.

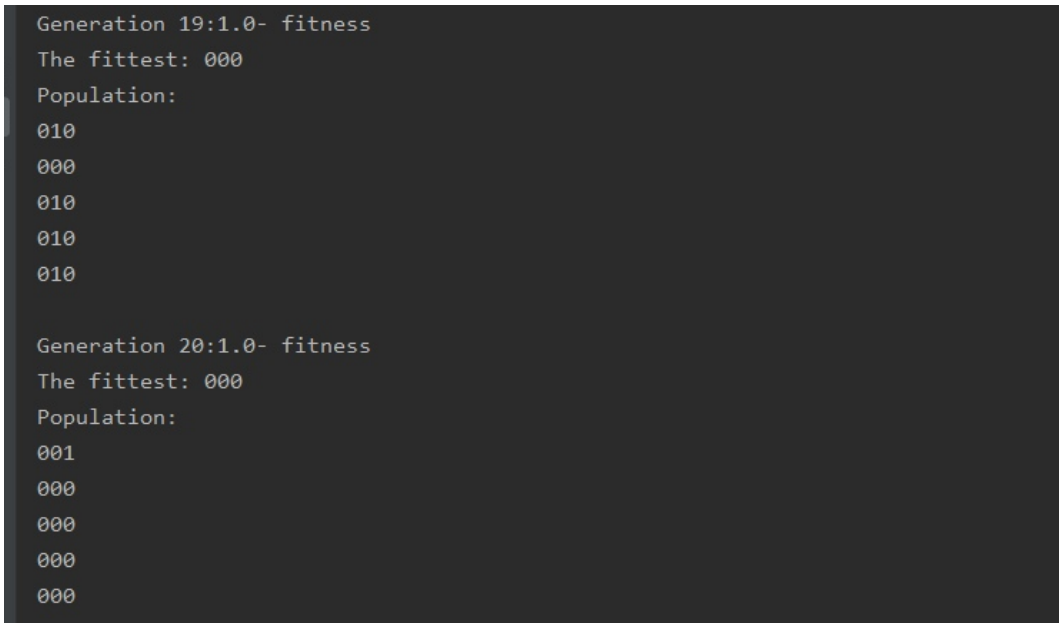

Рисунок 13: Результаты работы ГА генитор для функции (2)

Однако после повторного запуска алгоритма результаты были уже следующие:

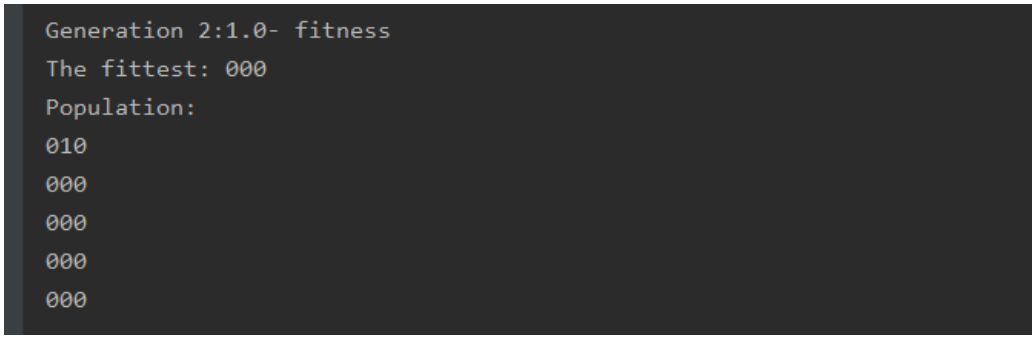

Рисунок 14: Результаты работы ГА генитор для функции (2)

В итоге оказалось, что оба алгоритма работают одинаково быстро, только генитор изредка может “тормозить”.

2.4.3. Результаты работы ГА для функции с синусом

Целевая функция будет иметь вид:

$f(x)=x^3+5\sin(x+3)$	(3)

Ее график выглядит следующим образом:

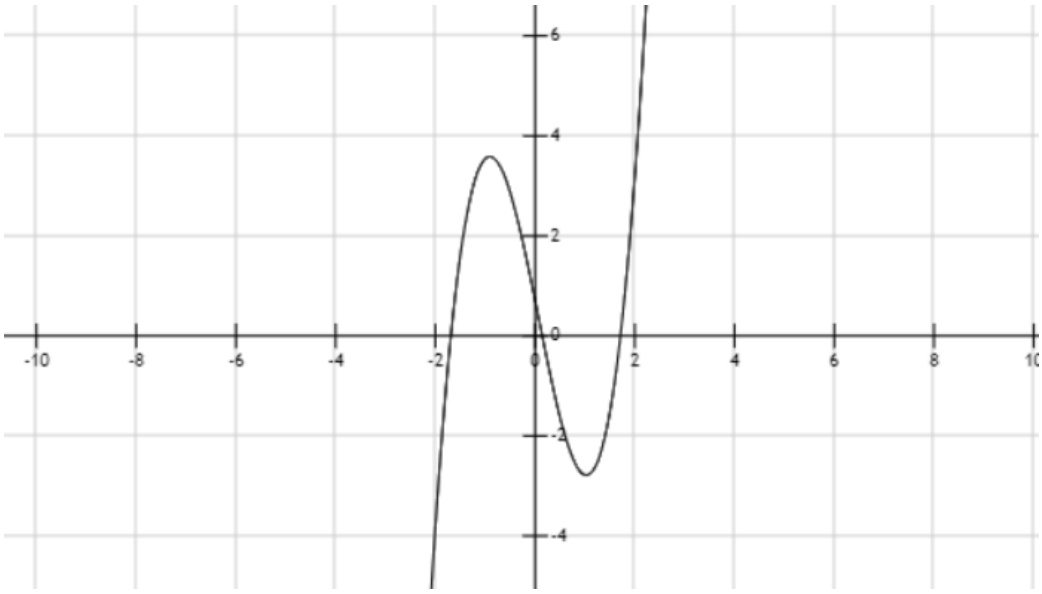

Рисунок 15: График функции (3)

Минимум данной функции находится приблизительно в точке 1(001).

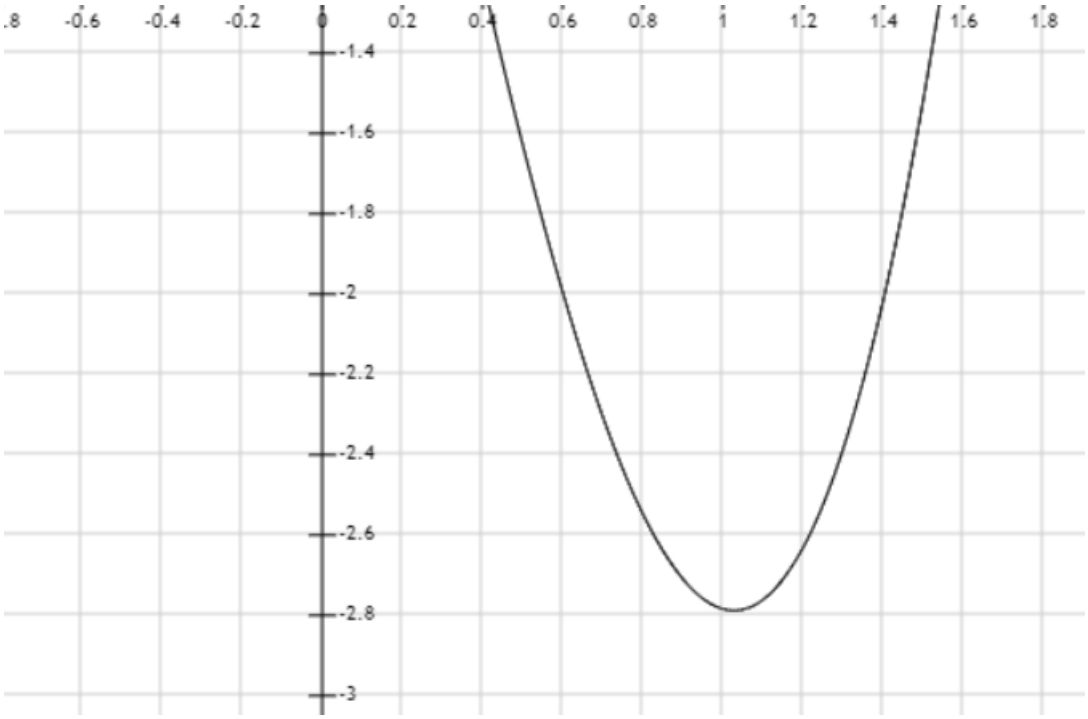

Рисунок 16: График функции (3)

Результаты работы классического генетического алгоритма для функции с синусом представлены на рис.17.

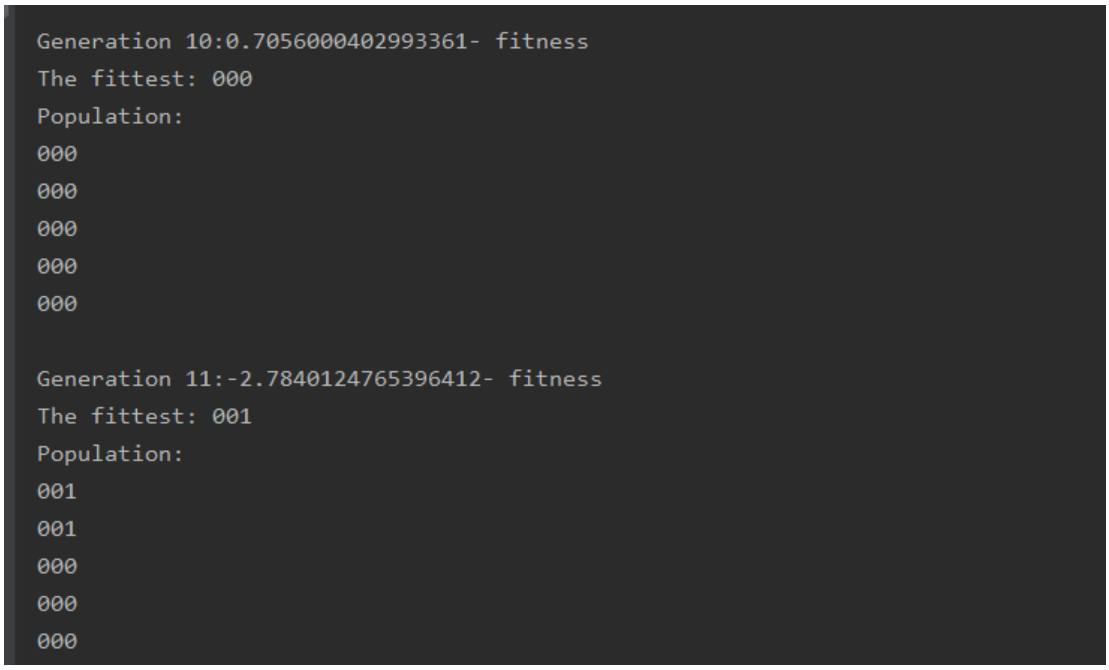

Рисунок 17: Результаты работы классического ГА для функции (3)

Как видно на рисунке, решение было найдено на 11 шаге цикла — это медленнее, чем было в случаях линейной и квадратичной функций.

Генитор определил глобальный минимум в два раза быстрее: решение появилось уже на 5 шаге.

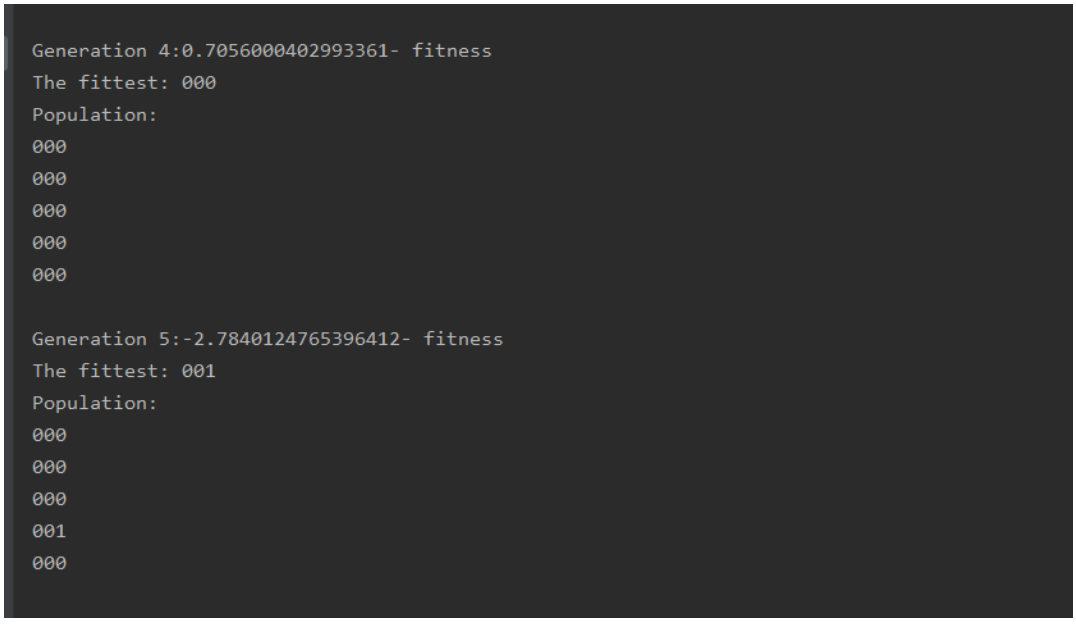

Рисунок 18: Результаты работы ГА генитор для функции (3)

После всех тестов можно подвести некоторый **итог**.

На диаграмме (рис.19) представлено сравнение скорости работы двух алгоритмов для линейной, квадратичной функций и функции с синусом.

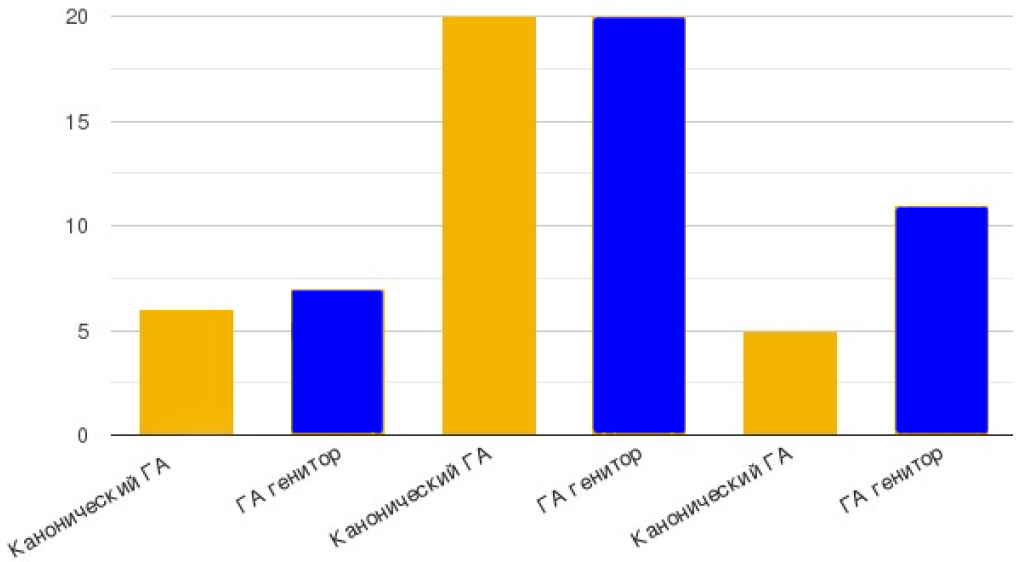

Рисунок 19: Диаграмма

Как я заметила, классический генетический алгоритм лучше всего применять в задачах, где целевая функция довольно простая (например, линейная). В то время как генитор успешнее всего в работе со сложными функциями - их глобальный минимум он вычисляет быстрее, чем классический ГА.

ЗАКЛЮЧЕНИЕ

На основании проделанного в ходе курсовой работы исследования можно сделать вывод, что генетические алгоритмы являются универсальным вычислительным средством для выполнения разнообразных оптимизационных задач, что подтверждает актуальность данной работы.

Отличительными характеристиками генетических алгоритмов являются:

- использование кодированных параметров, а не самих параметров;
- генетические алгоритмы находят популяцию наилучших решений, а не отдельное решение;
- генетические алгоритмы используют непосредственно значение целевой функции;
- генетические алгоритмы имеют множество различных модификаций, которые в одной задаче могут привести к неожиданному улучшению результатов, а в другой, наоборот, “тормозить” программу.

Генетические алгоритмы могут применятся в случаях, когда неизвестен способ точного решения задачи, либо когда способ существует, но слишком сложен в реализации. Для того, чтобы использовать генетический алгоритм, необходимо выполнить ряд действий:

1. Определить неизвестные переменные задачи.
2. Задать оценочную функцию приспособленности.
3. Выбрать способ кодирования переменных.
4. Определить параметры ГА.

В данной курсовой работе мною были рассмотрены два вида генетических алгоритмов: генитор и классический ГА. Проанализировав полученные в ходе выполнения практической части результаты, я пришла к выводу что генитор - более универсален, чем классический алгоритм, также он находит решение оптимизационной задачи быстрее. Однако классический ГА все же имеет право на существование: в ряде случаев рациональнее использовать именно его.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Вороновский Г.К., и др. В75 Генетические алгоритмы, искусственные нейронные сети и проблемы виртуальной реальности / Г. К. Вороновский, К. В. Махотило, С. Н. Петрашев, С. А. Сергеев.— Х.: ОСНОВА, 1997.— 112 с. ISBN 5—7768—0293—8.

2. Гладков Л.А., Курейчик В.В., Курейчик В.М. Генетические алгоритмы [Текст]/ Под ред. В.М. Курейчика. — 2-е изд., испр. и доп. — М.: ФИЗМАТЛИТ, 2010. — 368 с. — ISBN 978-9221-0510-1.

3. Генетические алгоритмы — http://qai.narod.ru/GA/gamodels.html

4. Генетический алгоритм — http://qai.narod.ru/Publications/tsoy_chapterGA.pdf

5. Панченко, Т. В. Генетические алгоритмы [Текст]: учебно-методическое пособие / под ред. Ю. Ю. Тарасевича. — Астрахань: Издательский дом «Астраханский университет», 2007. — 87 [3] с.

6. Что такое генетические алгоритмы. — https://www.itweek.ru/themes/detail.php?ID=51105.

ПРИЛОЖЕНИЕ

```
package by.bsu.args.run;

import by.bsu.args.algorithm.ClassicGeneticAlgorithm;
import by.bsu.args.algorithm.GenitorGeneticAlgorithm;
import by.bsu.args.entity.Constant;
import by.bsu.args.entity.Population;

public class Main {

    public static void main(String[] args) {
        GenitorGeneticAlgorithm algorithm = new GenitorGeneticAlgorithm();
        //ClassicGeneticAlgorithm algorithm = new ClassicGeneticAlgorithm();
        Population population = new Population(5);
        population.initialize();
        int generationCounter = 0;
        while (generationCounter != Constant.SIMULATION_LENGTH){
            ++generationCounter;
            System.out.println("\n"+"Generation " + generationCounter+":" +
population.getFittestIndividual().getFitness() + "- fitness" );
            System.out.println("The fittest:
"+population.getFittestIndividual().toString());
            System.out.println("Population:");
            for(int i=0; i<population.size();i++){
                System.out.println(population.getIndividual(i).toString());
            }
            population = algorithm.evolvePopulation(population);
        }
        System.out.println("The minimum is found!");
        System.out.println(population.getFittestIndividual());

    }
}

package by.bsu.args.entity;

public class Constant {
    private Constant(){}
    public static final double CROSSOVER_RATE = 0.05;
    public static final double MUTATION_RATE = 0.15;
    public static final int TOURNAMENT_SIZE = 5;
```

```
    public static final int CHROMOSOME_LENGTH = 3;
    public static final int GENE_LENGTH = 3;
    public static final int SIMULATION_LENGTH=100;

}


package by.bsu.args.entity;
import java.util.Random;

public class Chromosome {
    private int[] genes;
    private double fitness;
    private Random randomGenerator;

    public Chromosome(){
        this.genes = new int[Constant.CHROMOSOME_LENGTH];
        this.randomGenerator = new Random();
    }

    public void generateIndividual(){
        for (int i=0; i<Constant.CHROMOSOME_LENGTH;i++){
            int gene = randomGenerator.nextInt(2); // only 0 and 1
            genes[i]=gene;
        }
    }

    public double f(int x){
        //return 5-24*x+17*x*x-3.66666666667*x*x*x+0.25*x*x*x*x;
        //return x+2;
        //return 17*x*x + 3*x +1;
        return Math.sin(x+3)*5 + x*x*x;
    }
    public int genesToInt(){
        int base = 1;
        int geneInInt = 0;

        for(int i=0; i<Constant.GENE_LENGTH;++i){
            if(this.genes[i] == 1) {
                geneInInt += base;
            }

            base=base*2;
```

```
        }
        return geneInInt;
    }

    public double getFitness(){
       int genesToInt = genesToInt();
        return f(genesToInt);
    }

    public int getGene(int index){
        return this.genes[index];
    }

    public void setGene(int index, int value){
        this.genes[index] = value;
        this.fitness = 0;
    }

    @Override
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(int i =genes.length-1; i>=0;i--){
            result.append(genes[i]);
        }
        return result.toString();
    }
}


package by.bsu.args.entity;

public class Population {
    private Chromosome[] chromosomes;

    public Population(int populationSize){
        chromosomes = new Chromosome[populationSize];
    }

    public void initialize(){
        for(int i = 0; i< chromosomes.length; ++i){
            Chromosome newChromosome = new Chromosome();
            newChromosome.generateIndividual();
            saveIndividual(i, newChromosome);
```

```
        }
    }

    public Chromosome getIndividual(int index){
        return this.chromosomes[index];
    }

    public Chromosome getFittestIndividual(){
        Chromosome fittest = chromosomes[0];
        for(int i = 1; i< chromosomes.length; ++i){
            if(getIndividual(i).getFitness() < fittest.getFitness())
                fittest=getIndividual(i);
        }
        return fittest;
    }

    public int size(){
        return chromosomes.length;
    }
    public void saveIndividual(int index, Chromosome chromosome){
        this.chromosomes[index]= chromosome;
    }
}
```

```
package by.bsu.args.algorithm;

import by.bsu.args.entity.Chromosome;
import by.bsu.args.entity.Constant;
import by.bsu.args.entity.Population;
import java.util.Random;

public class ClassicGeneticAlgorithm {
    private Random randomGenerator;

    public ClassicGeneticAlgorithm(){
        this.randomGenerator = new Random();
    }

    public Population evolvePopulation(Population population){
        Population children = new Population(population.size());
        for(int i = 0; i<population.size(); i++){
            Chromosome firstChromosome = randomSelection(population);
```

```
            Chromosome secondChromosome = randomSelection(population);
            Chromosome newChromosome = crossover(firstChromosome,
secondChromosome);
            children.saveIndividual(i, newChromosome);
        }
        for(int i=0; i< children.size();++i){
            mutate(children.getIndividual(i));
        }
        Population newPopulation = new Population(4);
        newPopulation = selection(population, children);

        return newPopulation;
    }

    public Population selection(Population population, Population children){
        Population newPopulation = new
Population(Constant.TOURNAMENT_SIZE*2);
         for(int i=0; i< population.size();i++){
             newPopulation.saveIndividual(i,population.getIndividual(i));
         }
        for(int i=population.size(); i< newPopulation.size();i++){
            newPopulation.saveIndividual(i,children.getIndividual(i-
Constant.TOURNAMENT_SIZE));
        }
        sort(newPopulation);
        Population sortedPopulation = new Population(population.size());
        for(int i=0; i< population.size();i++){
            sortedPopulation.saveIndividual(i, newPopulation.getIndividual(i));
        }
        return sortedPopulation;
    }

    public void sort(Population population){
        boolean isSorted = false;
        Chromosome temp;
        while (!isSorted){
            isSorted = true;
            for(int i=0; i<population.size()-1;i++){

if(population.getIndividual(i).getFitness()>population.getIndividual(i+1).getFitness()
){
                    isSorted = false;
                    temp = population.getIndividual(i);
```

```
                population.saveIndividual(i,population.getIndividual(i+1));
                population.saveIndividual(i+1,temp);
            }
        }
    }

    public Chromosome randomSelection(Population population){
        int  randomIndex = (int) (Math.random()*population.size());
        return population.getIndividual(randomIndex);
    }

    public void mutate(Chromosome chromosome){
        for(int i = 0; i< Constant.CHROMOSOME_LENGTH; ++i){
            if(Math.random() <= Constant.MUTATION_RATE){
                int gene = randomGenerator.nextInt(2);
                chromosome.setGene(i,gene);
            }
        }
    }

    public Chromosome crossover(Chromosome firstChromosome, Chromosome
secondChromosome){
        Chromosome newSolution = new Chromosome();
        for(int i=0; i< Constant.CHROMOSOME_LENGTH; ++i){
            if(Math.random() <= Constant.CROSSOVER_RATE){
                newSolution.setGene(i, firstChromosome.getGene(i));
            } else{
                newSolution.setGene(i, secondChromosome.getGene(i));
            }
        }
        return newSolution;
    }
}


package by.bsu.args.algorithm;

import by.bsu.args.entity.Chromosome;
import by.bsu.args.entity.Constant;
import by.bsu.args.entity.Population;
import java.util.Random;
```

```
public class GenitorGeneticAlgorithm {
    private Random randomGenerator;

    public GenitorGeneticAlgorithm(){
        this.randomGenerator = new Random();
    }

    public Population evolvePopulation(Population population){
        for(int i = 0; i<population.size(); ++i){
            Chromosome firstChromosome = randomSelection(population);
            Chromosome secondChromosome = randomSelection(population);
            Chromosome newChromosome = crossover(firstChromosome,
secondChromosome);
            int index = findWeakest(population);
            population.saveIndividual(index, newChromosome);
        }
        for(int i=0; i< population.size();++i){
            mutate(population.getIndividual(i));
        }
        return population;
    }

    public int findWeakest(Population population){
        int temp=0;
        Chromosome min = population.getIndividual(0);
            for(int i=1; i<population.size();i++){
                if(population.getIndividual(i).getFitness()>min.getFitness()){
                    temp = i;
                    min=population.getIndividual(i);
                }
            }
        return temp;
    }

    public void mutate(Chromosome chromosome){
        for(int i = 0; i< Constant.CHROMOSOME_LENGTH; ++i){
            if(Math.random() <= Constant.MUTATION_RATE){
                int gene = randomGenerator.nextInt(2);
                chromosome.setGene(i,gene);
            }
        }
    }
```

```
    public Chromosome crossover(Chromosome firstChromosome, Chromosome
secondChromosome){
        Chromosome newSolution = new Chromosome();
        for(int i=0; i< Constant.CHROMOSOME_LENGTH; ++i){
          if(Math.random() <= Constant.CROSSOVER_RATE){
              newSolution.setGene(i, firstChromosome.getGene(i));
            }  else{
              newSolution.setGene(i, secondChromosome.getGene(i));
           }
        }
        return newSolution;
        }

    public Chromosome randomSelection(Population population){
        Population newPopulation = new Population(Constant.TOURNAMENT_SIZE);
        for(int i=0; i<Constant.TOURNAMENT_SIZE; ++i){
           int  randomIndex = (int) (Math.random()*population.size());
           newPopulation.saveIndividual(i, population.getIndividual(randomIndex));
        }
        Chromosome fittestChromosome = newPopulation.getFittestIndividual();
        return fittestChromosome;
    }
}
```