МИНОБРНАУКИ РОССИИ

Санкт-Петербургский государственный

электротехнический университет

«ЛЭТИ» им. В.И. Ульянова (Ленина)

Кафедра МО ЭВМ

ОТЧЁТ

по учебной практике

по направлению “Генетические алгоритмы”

Тема: **Задача о рюкзаке**

Преподаватель Жангиров Т. Р.

Санкт-Петербург

2025

Роли в бригаде

Боривец Савелий - руководитель, разработчик GUI

Канцеров Артемий - программист(реализация основного функционала)

Якушев Пётр - программист(ответственный за мутации и скрещивания)

Интерфейс

Окно ввода

Предлагается выбрать несколько способов ввода исходных данных:

- Случайная генерация
- Ввод из txt-файла
- Ввод из GUI

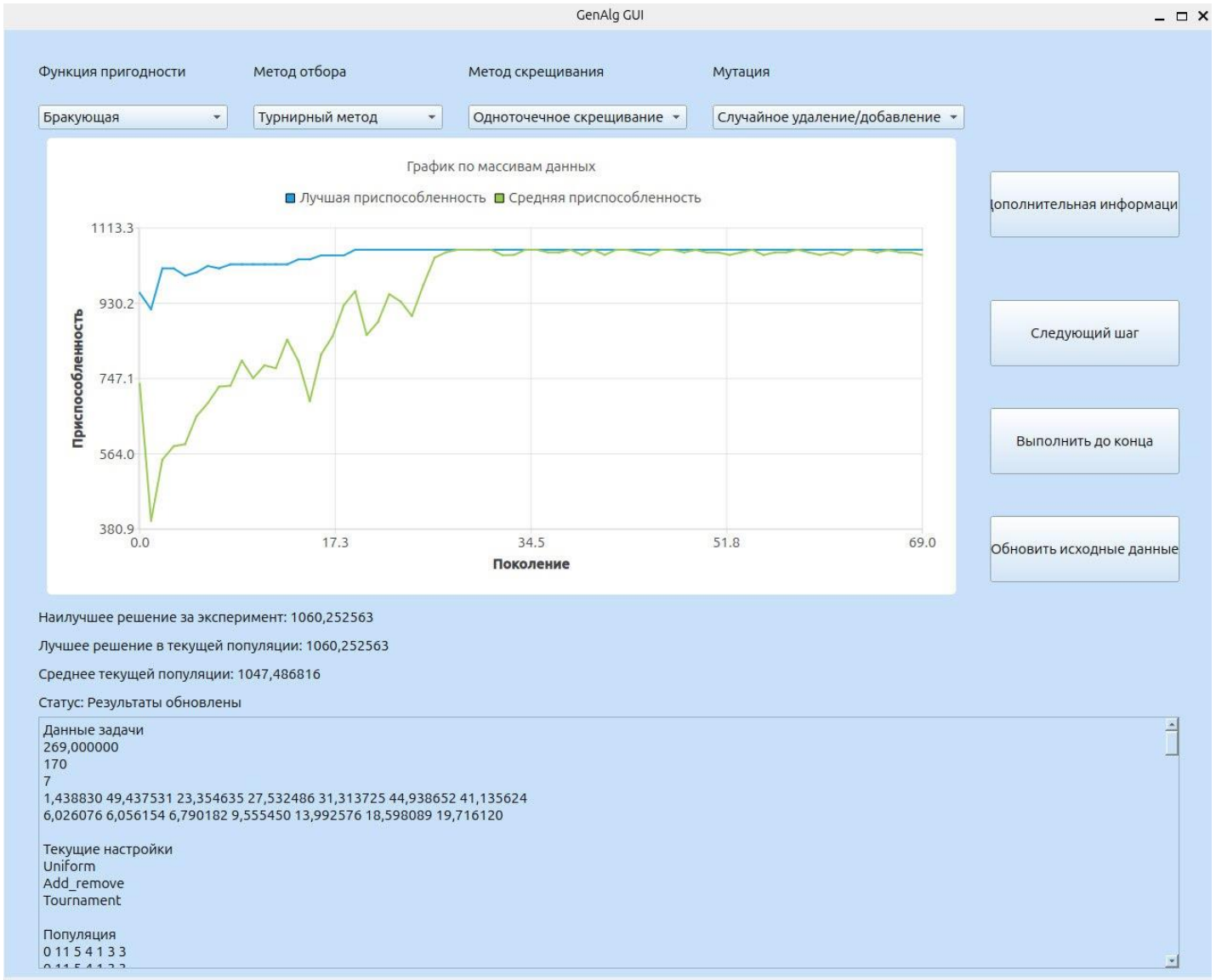
Если будут возникать проблемы, предупреждение снизу поможет разобраться, что не так.

Окно отслеживания работы генетического алгоритма

После того как были введены данные, алгоритм выполняет свою работу.

Можно выбрать различные функции пригодности, скрещивания, мутации и методы отбора, а также выбрать вероятность мутации и скрещивания. Доступно сохранение текущей популяции в txt-файл, выполнение следующего шага, выполнение до конца(до момента критерия окончания процесса), можно начать заново с ранее введенными данными, а также обновить ранее введенные параметры.

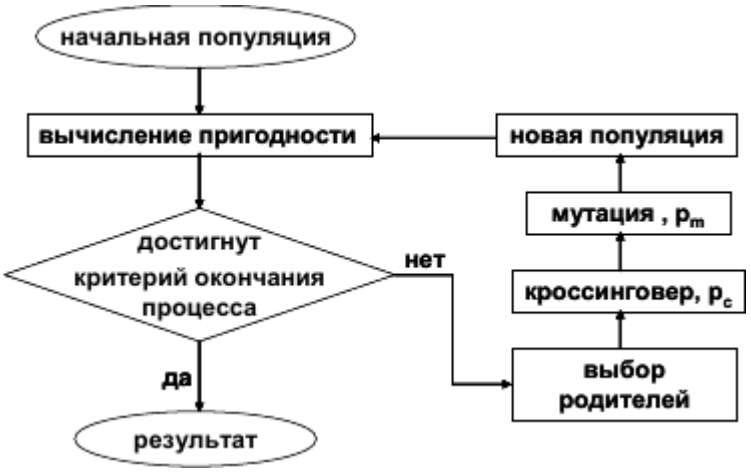
Для наглядности будут реализованы графики лучшей пригодности и средней пригодности популяции, а также данные о наилучшем решении и средней стоимости популяции.



План решения задачи

Для реализации генетического алгоритмы для решения задачи о рюкзаке необходимо определиться:

- Что будет хромосомой(особью)?
- Как будет работать функция вычисления пригодности?
- Какими методами будет проходить отбор родителей?
- Как будет работать скрещивание?
- Как будет работать мутация?



Хромосома

Список, представляющий из себя заполненный рюкзак, в который больше не положить любую вещь так, чтобы она не привела к переполнению рюкзака

Геном в хромосоме будет количество i-ого предмета в упорядоченном по весу списке вещей

Упорядоченный список вещей

Класс, который будет хранить список упорядоченных по весу предметов, содержать поле максимальной удельной стоимости(для “идеального рюкзака”). Добавляемые предметы будет вставлять на корректные места. Предметы с весом меньшим, либо равным нулю будет исключать

Критерий окончания алгоритма

Критерием окончания алгоритма будет несущественная разница между лучшими стоимостями поколения. Это означает то, что генетический алгоритм пришел к максимуму.

Функция вычисления пригодности

Функция вычисления пригодности(фитнесс-функция)— это функция, которая принимает на вводе потенциальное решение проблемы и выдаёт значение, оценивающее его пригодность. В случае задачи о рюкзаке таким значением будет являться общая стоимость вещей в рюкзаке.

Планируется реализовать следующие функции вычисления пригодности:

1. Если общий вес предметов меньше или равен максимальной вместимости рюкзака, то функция вычисления вероятности вернет общую стоимость предметов в рюкзаке. В ином случае она вернет 0, то есть “забракует” решение, из-за чего у него будет гораздо меньше или вообще не будет шансов попасть следующий отбор.
2. К общей стоимости добавляются лишь те предметы, которые при добавлении к текущему общему весу не приведут к превышению максимальной вместимости. Выбор таких предметов будет либо от меньшего к большему, либо случайным образом среди ненулевых элементов.

Метод отбора родителей

В качестве методов отбора планируется создать несколько методов, чтобы посмотреть, какой из них будет наиболее эффективным.

1. Метод отбора по правилу рулетки, или отбор пропорционально приспособленности

В *методе рулетки (roulette-wheel selection)* особи отбираются с помощью N «запусков» рулетки, где N — размер популяции. Колесо рулетки содержит по одному сектору для каждого члена популяции. Размер i-го сектора пропорционален вероятности попадания в новую популяцию $P(i)$, вычисляемой по формуле:

$$P(i) = \frac{f(i)}{\sum_{i=1}^{N} f(i)},$$

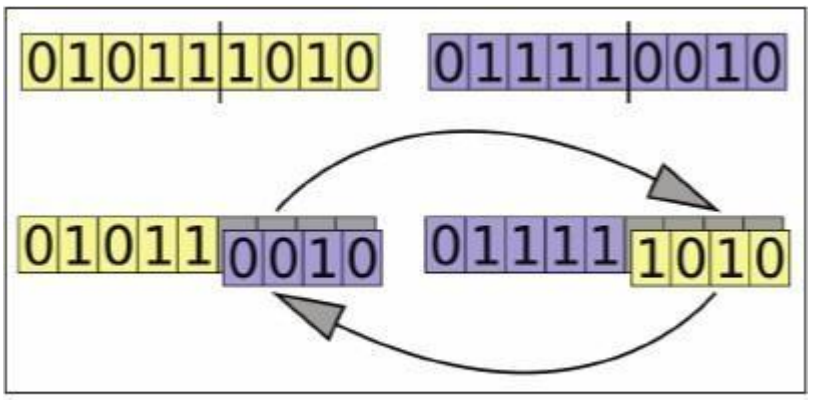
У особей с меньшей приспособленностью будет меньше шансов попасть в следующую популяцию для скрещивания.

2. Турнирный отбор - будет проводиться несколько раундов с фиксированным количеством участников. Среди участников для последующего скрещивания будет взят тот, у которого приспособленность оказалась больше всего. Количество раундов равно размеру популяции. Чем больше участников турнира, тем выше шанс того, что в раунде будут появляться лучшие особи.

Методы скрещивания

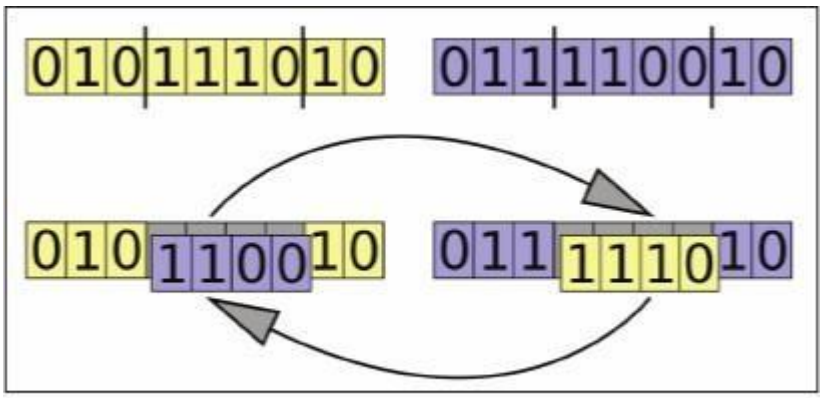
В качестве скрещивания планируется реализовать следующие методы:

1. Одноточечное скрещивание - в хромосоме случайно выбирается позиция - точка скрещивания, затем гены одной хромосомы, расположенные справа от точки скрещивания обмениваются с генами второй хромосомы, расположенные справа от точки скрещивания



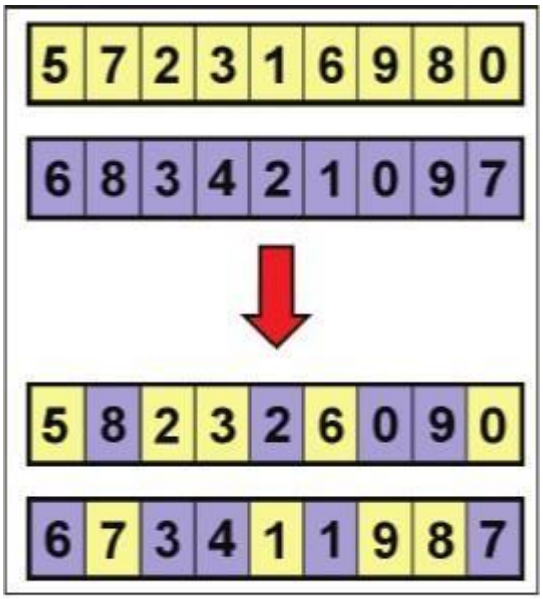
Пример одноточечного скрещивания

2. Двухточечное скрещивание - метод похож на предыдущий, но теперь выбираются две случайные точки скрещивания, и гены хромосом заключенные между этими точками обмениваются друг с другом



Пример двухточечного скрещивания

3. Равномерное скрещивание - с вероятностью 50% гены обмениваются хромосомами



Пример равномерного скрещивания

Методы мутации

Мутация – последний генетический оператор, применяемый при создании

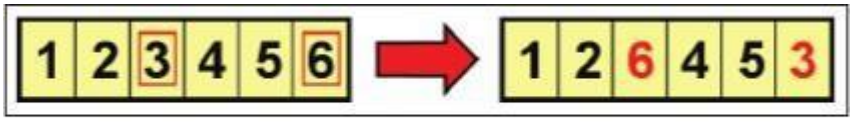
нового поколения.

Мутация будет выполняться с вероятностью 0-2%

Планируется реализовать следующие методы мутации:

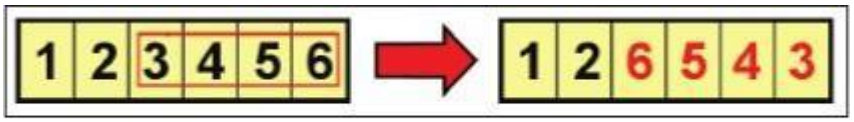
1. Случайное добавление или удаление одной вещи в рюкзак. Если ген будет равен 0(вещи данного типа 0), то вещь будет только добавляться. В ином случае будет выполняться добавление с вероятностью 50%, а в ином случае удаление

2. Мутация обменом - выбирается две позиции, целочисленные значения на данных позициях обмениваются



Пример мутации обменом

3. Мутация обращением - выбирается последовательность в хромосоме, порядок генов в последовательности изменяется на обратный



Пример мутации обращением

План реализации алгоритма

Для начала необходимо реализовать хранение данных, полученных из файла, введенных из GUI или случайной генерации. За это будет отвечать класс DataManager, у которого будут поля-данные, требуемые для решения задачи:

- Количество предметов
- Отсортированный в порядке возрастания веса вектор предметов-структур(информация о предметах будет в виде структур с тремя полями - ценой, весом и удельной стоимостью)
- Максимальная вместимость рюкзака

Данный класс будет иметь следующие методы:

- Метод, добавляющий предмет в список(будет вставлять на нужную позицию)
- Метод, который будет доставать случайный предмет с весом не превышающим значение в аргументах. Он будет нужен для случайного заполнения рюкзака при создании начальной популяции
- Метод, читающий txt-файлы, в которых будет находиться исходные данные задачи
- Сеттеры, которые будут устанавливать полученные данные
- Метод, генерирующий случайные данные задачи

Для хранения данных о хромосоме(особи) будет использоваться класс Backpack с полями:

- Вектор с количеством предметов в рюкзаке(предмет на i-ой позиции будет иметь цену и вес предмета на i-ой позиции в списке DataManager)
- Поле, отвечающее за общий вес текущего рюкзака
- Поле, отвечающее за общую цену текущего рюкзака У него будут методы:
- Добавление i-ого предмета в рюкзак
- Удаление i-ого предмета

Функции пригодности будут реализованы в отдельном классе. Главный метод будет вызывать вспомогательный метод,

возвращающий пригодность решения. Пользователь сможет выбрать функцию пригодности.

Мутации будут реализованы в отдельном классе. Главный метод будет выполнять мутацию с определенной вероятностью, вызывая вспомогательные методы, которые будут принимать объект класса Backpack, выполнять операции мутации, а затем возвращать измененный(мутировавший) объект класса Backpack. Пользователь сможет выбрать вид мутации, а также ее вероятность.

Будет реализован отдельный класс под скрещивания. Главный метод будет выполнять скрещивание с определенной вероятностью, вызывая вспомогательные методы, которые будут принимать два объекта класса Backpack, выполнять операции скрещивания, а затем возвращать два потомка скрещиваемых классов Backpack.

Пользователь сможет выбрать вид скрещивания, а также вероятность скрещивания.

Работой алгоритма(взаимодействия между классами) будет управлять отдельный класс GenAlg.

Он будет:

1. Генерировать начальную популяцию из n хромосом.
2. Вычислять для каждой хромосомы ее пригодность.
3. Выбираем пару хромосом-родителей с помощью одного из способов отбора.
4. Проводить скрещивание двух родителей с заданной вероятностью, производя двух потомков.
5. Выполнять мутацию с заданной вероятностью
6. Повторять шаги 3–5, пока не будет сгенерировано новое поколение популяции, содержащее n хромосом.
7. Повторяем шаги 2–6, пока не будет достигнут критерий окончания процесса.

Для графического интерфейса(GUI) будет использоваться библиотека Qt.

Описание работы созданных классов

DataManager

Класс содержит поля для максимальной вместимости, размера популяции отсортированных по возрастанию веса предметов и типа функции пригодности (мягкая/жёсткая), которая будет использоваться позже в других классах и позволять задавать строгость вычисления пригодности популяции

```
std::vector<Item> items; Предметы
float maxCapacity; Вместимость
int populationSize; Популяция
FitnessType fitness; Строгость пригодности
```

`enum` класс, который помогает задать строгость вычисления пригодности

```
enum class FitnessType {
  Cutting,Жёсткий метод
  Gentle Мягкий метод
};
```

Вектор предметов сделан с помощью структуры `Item`, содержащей информацию о цене предмета, его весе и отношении этих значений

```
struct Item
{
    double price; Цена
    double weight; Вес
    double unitPrice; Отношение веса к цене
};
```

Класс содержит ряд методов. Среди них метод `binarySearch`, который реализует алгоритм бинарного поиска, для нахождения оптимального места вставки номого элемента

```
int DataManager::binarySearch(float weight){
  int left = 0;
  int right = items.size();
  while (left < right) {
    int mid = left + (right - left) / 2;
    if (items[mid].weight <= weight) {
      left = mid + 1;
    } else {
      right = mid;
    }
```

```
    }
    return left;
  }
```

Непосредственно сам метод вставки нового элемента

```
  void DataManager::add(float price, float weight){
    if (weight <= 0 || weight > maxCapacity || price
<= 0) {
      return; Если данные некорректны, идём дальше
    }
    Item something; Новый элемент
    something.price = price; Присвоение цены
    something.weight = weight; Присвоение веса
    something.unitPrice = price / weight;Присвоение
отношения
    int pos = binarySearch(weight); // Поиск места
    items.insert(items.begin() + pos, something);
  }
```

Метод `randomTake` выбирает случайный элемент, чей вес не превышает заданной границы, и возвращает его индекс

```
  int DataManager::randomTake(float upBoard) {
    std::random_device rd;
    std::mt19937 gen(rd());
    int pos = binarySearch(upBoard); Поиск элемента с
таким же весом
    if (pos == 0) {
      return -1; Если не нашлось подходящего
элемента, значит ничего не можем положить в рюкзак
    }
    std::uniform_int_distribution<> dist(0, pos - 1);
    return dist(gen); Возврат случайного индекса
  }
```

Возможность загрузить данные из файла реализована в методе `loadFile`.

```
     std::string DataManager::loadFile(std::string
path) {
      if (path.empty()) return "Введите путь до txt
файла";

      std::ifstream file(path); Путь до файла
      if (!file.is_open()) {
        return "Ошибка: Файл " + path + " не
получилось открыть";
      }

    Загрузка содержимого в одну строку
      std::string fileText(
          (std::istreambuf_iterator<char>(file)),
          std::istreambuf_iterator<char>()
      );

      std::string parseStatus =
stringParse(fileText);

      file.close();

      return parseStatus;
    }
```

Данный метод позволяет правильно считать данные из файла

```
    std::string DataManager::stringParse(std::string
input) {
      if (input.empty()) return "Пустой файл/окно
ввода";

      std::istringstream iss(input);

      int itemsNum;
      iss >> maxCapacity >> populationSize >>
itemsNum;

      if (iss.fail()) {
        return "Ошибка: Основные параметры не
прочитаны";
      }

      Цены
      std::vector<double> prices(itemsNum);
      for (int i = 0; i < itemsNum; ++i) {
        if (!(iss >> prices[i])) {
          return "Ошибка: Некорректные цены";
        }
      }

      Веса
      std::vector<double> weights(itemsNum);
      for (int i = 0; i < itemsNum; ++i) {
        if (!(iss >> weights[i])) {
          return "Ошибка: Некорректные веса";
        }
      }

      Проверяем, что все данные считаны корректно
      if (!iss.eof()) {
        std::string remaining;
        std::getline(iss, remaining);
        if (!remaining.empty()) {
          return "Предупреждение: лишние данные в
```

```
конце строки: " + remaining;
        }
      }
      for (int i = 0; i < itemsNum; ++i) {
        add(prices[i], weights[i]);
      }

      return "Данные успешно считаны";
    }
```

Пользователь имеет возможность сгенерировать случайные значения. Для этого существует соответствующий метод

```
    void DataManager::randomLoad() {
      std::random_device rd;
      std::mt19937 gen(rd());
      std::uniform_int_distribution<>
capacityDist(50, 1000);
      maxCapacity = capacityDist(gen); Случайная
вместимость
      std::uniform_int_distribution<>
populationDist(50, 300);
      populationSize = populationDist(gen); Случайный
размер популяции
      std::uniform_int_distribution<> count(4, 15);
      int numItems = count(gen); Случайное количество
предметов

      std::uniform_real_distribution<>
weightDist(1.0, 20.0);
      std::uniform_real_distribution<> priceDist(1.0,
50.0);

    Случайные значения веса, стоимости, их отношения
для каждого предмета
      float price, weight;
      for (int i = 0; i < numItems; ++i) {
        price = priceDist(gen);
        weight = weightDist(gen);
        add(price, weight);
      }
    }
```

В классе также присутствуют методы-геттеры, которые возвращают значения соответствубщих полей

```
    std::vector<Item> DataManager::getItems() {
      return items; Возвращает список предметов
    }

    float DataManager::getMaxCapacity() {
      return maxCapacity; Возвращает максимальную
вместимость
    }

    int DataManager::getPopulationSize() {
      return populationSize; Возвращает размер
популяции
    }

    int DataManager::getItemsNum() {
      return items.size(); Возвращает количество
предметов
    }

    void DataManager::setFitness(FitnessType fit) {
      fitness = fit; Устанавливает строгость
вычисления пригодности
    }

    FitnessType DataManager::getFitness() {
      return fitness; Возвращает строгость
пригодности
    }
```

Backpack

Класс рюкзака содержит всего одно поле - множество целых чисел, где каждый i-тый элемент является количеством соответствующего предмета в рюкзаке

```
    std::vector<int> solution; Количество предметов каждого типа
```

Подсчёт `solution` происходит с помощью следующих методов

```
Удаляет один предмет соответсвующего типа из рюкзака
void Backpack::delItem(int pos, DataManager data) {
  if (solution[pos] == 0) {
    return;
  }

  solution[pos] -= 1;
}

Добавляет новый предмет
void Backpack::addItem(int pos, DataManager data) {
  solution[pos] += 1;
}

Изменяет количество соответствующих предметов на конкретное число
void Backpack::editSolution(int pos, int amount) {
  solution[pos] = amount;
}
```

Предметы можно получить с помощью геттера

```
    std::vector<int> Backpack::getSolution() const {
      return solution; Возвращает количество каждого предмета
    }
```

В классе также присутствуют методы, позволяющие определить пригодность рюкзака, для оценивания возможности его дальнейшего скрещивания. Их отличает строгость выполнения. `getFitnessValue1` позволяет перегрузить рюкзак, не беря в таком случае рассматриваемый предмет, когда как `getFitnessValue2` сразу возвращает 0

```
    float Backpack::getFitnessValue1(DataManager data)
const {
      float totalVal = 0, weight = 0;
      std::vector<Item> items = data.getItems();

    Просмотр всех предметов
      for (size_t i = 0; i < items.size(); i++) {
    Если предмет можно добавить без превышения
ограничения по весу, он добавляется
        if (weight + items[i].weight * solution[i] <=
data.getMaxCapacity()) {
          totalVal += items[i].price * solution[i];
          weight += items[i].weight * solution[i];
        }
      }
      return totalVal; Возвращает пригодность
    }

    float Backpack::getFitnessValue2(DataManager data)
const {
      float totalVal = 0, weight = 0;
      std::vector<Item> items = data.getItems();

    Просмотр всех предметов
      for (size_t i = 0; i < items.size(); i++) {
        totalVal += items[i].price * solution[i];
        weight += items[i].weight * solution[i];
    Возвращает 0 при превышении лимита
        if (weight > data.getMaxCapacity()) {
          return 0;
        }
      }
      return totalVal; Возвращает пригодность

    }
```

Направить в правильной метод вычисления пригодности помогает метод `getFitnessValue`, который использует для этого класс `FitnessType`, разобранный ранее

```
    float Backpack::getFitnessValue(FitnessType fitness,
DataManager data) const {
      float totalVal; Значение пригодности
      if (fitness == FitnessType::Gentle) {
        totalVal = getFitnessValue1(data); Мягкое
вычисление
      } else {
        totalVal = getFitnessValue2(data); Строгое
вычисление
      }
      return totalVal;
    }
```

Mutation

Класс позволяет работать с мутациями. Чтобы их можно было выбрать, в дополнение к `Mutation` был создан `enum` класс `MutationType`, который содержит все возможные мутации

```
    enum class MutationType {
        ADD_REMOVE, Удаление/вставка
        CHANGE Обмен
    };
```

Полей у класса два. Одно, приватное, хранит выбранную мутацию, когда как второе уже может быть изменено извне, и содержит вероятность получения мутации

```
    MutationType mutation; Выбранный тип мутации
    float IsMutation; Вероятность применения мутации
```

Получение мутации происходит с некоторой вероятностью. Если было принято решение совершить мутацию, вызывается один из мутационных методов.

```
    void Mutations::getMutation(Backpack& backpack) {
      std::uniform_real_distribution<float>
chance(0.0, 1.0); Случайное float число от 0 до 1

    Если это случайное число превышает вероятность
получения мутации, то мутации не будет
      if (chance(gen) > IsMutation) return;
std::vector<int> solution = backpack.getSolution();

    Выбор метода для получения мутации
    switch (mutation) {
        case MutationType::ADD_REMOVE:
          mutateAddOrRemove(backpack);
          break;
        case MutationType::CHANGE:
          mutateChange(backpack);
          break;
      }
    }
```

Пользователь может задать тип мутации вручную

```
    void Mutation::setType(MutationType t) {
      mutation = t; Установка желаемого типа
    }
```

Метод обмена производит взаимную замену количества двух предметов

```
    void Mutations::mutateChange(Backpack& backpack)
{
      std::vector<int> items =
backpack.getSolution();

    Если предметов меньше двух, то обмен невозможен
      if (items.size() < 2) return;
std::uniform_int_distribution<int> ind(0, items.size()
- 1);

    Выбор индексов для обмена
      int index1 = ind(gen);
      int index2 = ind(gen);
      while (index1 == index2) {
        index2 = ind(gen); Если индексы совпали
      }

    Присваивание новых значений
      backpack.editSolution(index1, items[index2]);
      backpack.editSolution(index2, items[index1]);
    }
```

Метод добавить/удалить изменяет количество выбранного предмета на 1

```
    void Mutations::mutateAddOrRemove(Backpack&
backpack) {
      std::vector<int> items =
backpack.getSolution();
      if (items.empty()) return;
      std::uniform_int_distribution<int> mut(0, 1);
      int way = mut(gen); В зависимости от значения,
будет понятно, какая группа должна лишиться/получить
предмет
      std::uniform_int_distribution<int> ind(0,
```

```
items.size() - 1);
      int index = ind(gen); Выбор индекса для обмена

    Добавление/удаление предмета
      if (way == 0) {
        if (items[index] == 0) { Если количество
предметов данного класса - 0,то удаление заменяется на
добавление
          backpack.editSolution(index, items[index] +
1);
          return;
        }
        backpack.editSolution(index, items[index] -
1);
      } else {
        backpack.editSolution(index, items[index] +
1);
      }
    }
```

Crossover

Класс содержит поля для вероятности проведения скрещивания и его типа

```
        float probability; Вероятность скрещивания
        CrossoverType type; Тип скрещивания
```

Как и класс `Mutation`, `Crossover` содержит `enum` класс-помощник `CrossoverType`, который позволяет определять тип скрещивания

```
    enum class CrossoverType {
        OnePoint, Одноточечное
        TwoPoint, Двухточечное
        Uniform  Равномерное
    };
```

В определении типа скрещивания также помогает метод `cross`, который, если с некоторой вероятностью было принято решение проводить скрещивание, распределяет родителей по соответствующим методам

```
    std::pair<Backpack, Backpack>
Crossover::cross(const Backpack& parent1, const
Backpack& parent2) {
      std::uniform_real_distribution<float>
chance(0.0, 1.0);

    Если особи не скрещиваются, они возвращаются без
изменений
      if (chance(gen) > probability)
        return {parent1, parent2};

    Выдача особям типа скрещивания
      switch (type) {
        case CrossoverType::OnePoint:
          return onePoint(parent1, parent2);
        case CrossoverType::TwoPoint:
          return twoPoint(parent1, parent2);
        case CrossoverType::Uniform:
          return uniform(parent1, parent2);
      }
    }
```

Тип скрещивания `onePoint` выбирает точку, по которой будет производиться обмен количеством вещей между особями и меняет их

```
    std::pair<Backpack, Backpack>
Crossover::onePoint(const Backpack& parent1, const
Backpack& parent2) {
      const auto& sol1 = parent1.getSolution();
      const auto& sol2 = parent2.getSolution();
      int size = sol1.size();
      std::uniform_int_distribution<int> dist(1, size
- 1);
      int point = dist(gen); Случайная точка деления
      std::vector<int> solution1(size, 0),
solution2(size, 0);

    Обмен между родителями
      for (int i = 0; i < size; i++) {
        if (i < point) {
          solution1[i] = sol1[i];
          solution2[i] = sol2[i];
        }
        else {
          solution1[i] = sol2[i];
          solution2[i] = sol1[i];
        }
      }

    Производство и возврат детей
      Backpack child1(solution1), child2(solution2);
      return {child1, child2};
    }
```

Тип скрещивания `twoPoint` выбирает отрезок внутри каждого родителя, который будет заменён на такой же отрезок другого родителя, и меняет их

```
    std::pair<Backpack, Backpack>
Crossover::twoPoint(const Backpack& parent1, const
Backpack& parent2) {
      const auto& sol1 = parent1.getSolution();
      const auto& sol2 = parent2.getSolution();
      int size = sol1.size();
      std::uniform_int_distribution<int> dist(0, size
- 1);
      int point1 = dist(gen); Начало отрезка
      int point2 = dist(gen); Конец отрезка

    Если конец меньше начала, замена
      if (point1 > point2) std::swap(point1, point2);
      std::vector<int> solution1(size, 0),
solution2(size, 0);

    Обмен между родителями
      for (int i = 0; i < size; i++) {
        if (i >= point1 && i <= point2) {
          solution1[i] = sol1[i];
          solution2[i] = sol2[i];
        }
        else {
          solution1[i] = sol2[i];
          solution2[i] = sol1[i];
        }
      }

    Производство и возврат детей
      Backpack child1(solution1), child2(solution2);
      return {child1, child2};
    }
```

Метод uniform перебирает каждый набор вещей каждого родителя и с вероятностью 50% обменивает их

```
    std::pair<Backpack, Backpack>
Crossover::uniform(const Backpack& parent1, const
Backpack& parent2) {
      const auto& sol1 = parent1.getSolution();
      const auto& sol2 = parent2.getSolution();
      int size = sol1.size();
      std::vector<int> solution1(size, 0),
solution2(size, 0);

    Монетка, которая будет давать 50% вероятность
обмена
      std::uniform_int_distribution<int> coin(0, 1);

    Обмен между родителями
      for (int i = 0; i < size; i++) {
        if (coin(gen) == 0) {
          solution1[i] = sol1[i];
          solution2[i] = sol2[i];
        }
        else {
          solution1[i] = sol2[i];
          solution2[i] = sol1[i];
        }
      }

    Производство и возврат детей
      Backpack child1(solution1), child2(solution2);
      return {child1, child2};
    }
```

Способ скрещивания можно установить вручную, с помощью соответствующего метода

```
    void Crossover::setType(CrossoverType t) {
      type = t; Тип скрешивания, заданный
пользователем
    }
```

GeneticAlgorithm

Класс реализует генетический алгоритм. К его полям относятся:

```
        DataManager Data; Набор предметов
        vector<Backpack> population; Популяция
        vector<float> fitnesses; Пригодности
        Crossover crossover; Класс скрещивания
        Mutations mutation; Класс мутаций
        SelectionType selectionMethod; Метод отбора
        vector<float> averageFitnessHistory; Средняя
пригодность по поколениям
        vector<float> bestFitnessHistory; Лучшая
пригодность по поколениям
        float bestCostOfAllTime; Лучшая стоимость
        int generationCount = 0; Счётчик поколений
        int maxGenerations; Критерий остановки
```

Также присутствует `enum` класс `SelectionType`, который позволяет определить тип отбора будущих родителей

```
    enum class SelectionType {
        Tournament, Отбор турниром
        Roulette Отбор рулеткой
    };
```

Алгоритм вызывается с помощью метода `run`. Метод итеративно запускает `runGeneration`, пока счётчик поколений не достигнет критерия остановки

```
    void GeneticAlgorithm::runGeneration() {
        Считаем приспособленность
        evaluateFitness();

averageFitnessHistory.push_back(AverageFitness());
        bestFitnessHistory.push_back(BestFitness());

        Новое поколение
        std::vector<Backpack> newPopulation;
        while (newPopulation.size() <
population.size()) {
            // Выбор родителей и их скрещивание
            auto [p1, p2] = selectParents();
            auto [c1, c2] = crossover.cross(p1, p2);
```

```
            Получение мутаций
            mutation.getMutation(c1);
            mutation.getMutation(c2);

            Добавление детей в новое поколение
            newPopulation.push_back(c1);
            if (newPopulation.size() <
population.size()){
                newPopulation.push_back(c2);
            }

        }
        population = std::move(newPopulation); Смена
поколений

    Вывод
        std::cout << "Лучший: " <<
bestFitnessHistory.back() << " Средний: " <<
averageFitnessHistory.back() << std::endl;
        for (int i = 0; i < 15; ++i) {
            std::cout <<
population[i].getFitnessValue(Data.getFitness(), Data)
<< ' ';
        }
        std::cout << std::endl;
        generationCount++;
    }
```

Сам метод run. Будет множество раз менять поколения, пока лучшее решение не повторится 50 раз подряд. Это будет означать, что решение невозможно улучшить

```
    void GeneticAlgorithm::run() {
        int count = 0;
        float previousFitness; Предыдущее лучшее
решение
        runGeneration();
        while (count != 50) {
            previousFitness =
bestFitnessHistory.back(); Обновление лучшего решения
            runGeneration(); Получение нового
поколения
```

```
            if (previousFitness ==
bestFitnessHistory.back()) {
                count++;
            } else {
                count = 0;
            }
        }

    Вывод
        std::cout << "Прогресс среднего: " <<
averageFitnessHistory.back() -
averageFitnessHistory[0]
            << " Прогрес лучшего: " <<
bestFitnessHistory.back() - bestFitnessHistory[0] <<
std::endl;
    }
```

Алгоритм можно перезапускать, просто введя новые значения

```
    void GeneticAlgorithm::restart(DataManager data)
{
        fitnesses.clear();
        averageFitnessHistory.clear();
        bestFitnessHistory.clear();
        generationCount = 0;
        for (int i = 0; i < Data.getPopulationSize();
++i) {
            Backpack randomBackpack(data);
            population.emplace_back(randomBackpack);
        }
    }
```

Приспособленность высчитывается с помощью метода `evaluateFitness`

```
    void GeneticAlgorithm::evaluateFitness() {
        fitnesses.clear();

         Высчитывание приспособленности для каждой
особи в популяции
        for (auto& indiv : population) {

fitnesses.push_back(indiv.getValueV1(Data));
```

```
            }
        }
```

Метод отбора турниром - один из способов в программе определить родителей, которые дадут новое поколение

```
    Backpack GeneticAlgorithm::tournamentSelection() {
        std::uniform_int_distribution<int> dist(0,
population.size() - 1);
        size_t n = 4;

        Выбор n уникальных случайных индексов
        std::vector<int> indices;
        while (indices.size() < n) {
            int idx = dist(gen);
            if (std::find(indices.begin(), indices.end(),
idx) == indices.end()) {
                indices.push_back(idx);
            }
        }

        Получение особи с максимальной приспособленностью
        Backpack* best = &population[indices[0]];
        for (size_t i = 1; i < n; ++i) {
            if
(population[indices[i]].getFitnessValue(Data.getFitness()
, Data) > best->getFitnessValue(Data.getFitness(), Data))
{
                best = &population[indices[i]];
            }
        }
        return *best;
    }
```

Кроме метода отбора турниром, существует метод отбора рулеткой

```
    Backpack GeneticAlgorithm::rouletteSelection() {

        Общая приспособленность
        float totalFitness = 0;
        for (auto& b : population){
            totalFitness += b.getValueV1(Data);
        }

        Взятие случайной длины
        uniform_real_distribution<float> dist(0,
```

```
totalFitness);
        float pick = dist(gen); Случайное число от 0 до
общей приспособленности
        float current = 0;

        Выбор особи
        for (auto& b : population) {
            current += b.getValueV1(Data);

        Если вышли за пределы длинны, то возвращаем
последний рассмотренный рюкзак
            if (current >= pick){
                return b;
            }
        }
        return population.back(); Если особь не была
выбрана
    }
```

Правильно распределить особи по методам отбора помогает метод selectParents

```
    pair<Backpack, Backpack>
GeneticAlgorithm::selectParents() {
        if (selectionMethod == SelectionType::Tournament)
{
            return {tournamentSelection(),
tournamentSelection()}; Отбор методом турнира
        } else {
            return {rouletteSelection(),
rouletteSelection()}; Отбор методом рулетки
        }
    }
```

Также в классе присутствуют различные геттеры и методы вычисления лучшей и средней приспособленности

```
    Вычисление средней приспособленности
    float GeneticAlgorithm::AverageFitness() {
```

```
        float sum = 0;
        for (auto& b : population)
            sum += b.getValueV1(Data);
        return sum / population.size();
    }


    Вычисление лучшей приспособленности
    float GeneticAlgorithm::BestFitness() {
        float best = 0;
        for (auto& b : population)
            best = max(best, b.getValueV1(Data));
        return best;
    }

    Передача текущей итерации
    int GeneticAlgorithm::get_genCount() {
        return generationCount;
    }

    Передача средней приспособленности
    vector<float> GeneticAlgorithm::get_averageFitness()
{
        return averageFitnessHistory;
    }

    Передача лучшей приспособленности
    vector<float> GeneticAlgorithm::get_bestFitness() {
        return bestFitnessHistory;
    }
```

Через `GeneticAlgorithm` происходит обращение к ранее рассмотренным классам, для задачи ряда значений вручную

```
    void
GeneticAlgorithm::setCrossoverType(CrossoverType t) {
      crossover.setType(t); Установка типа
скрещивания
    }

    void
GeneticAlgorithm::setMutationType(MutationType t) {
```

```
        mutation.setType(t); Установка типа мутации
    }

    void
GeneticAlgorithm::setSelectionType(SelectionType t) {
        selectionMethod = t; Установка метода отбора
    }
```

main функция

Основная функция программы, запускающая алгоритм для случайных входных данных

```
    int main() {
      DataManager dm;
      dm.randomLoad();
      dm.setFitness(FitnessType::Cutting);
      GeneticAlgorithm go(dm);
      go.run();
    }
```

Пример файла, подающегося в программу

```
200 100 6  Максимальная вместимость рюкзака, размер популяции и количество предметов соответственно
10.5 20.0 15.3 25.7 11.7 1  Цены предметов
3.2 5.1 4.8 6.0 2.9 5 Веса предметов
```

Описание работы GUI, его связь с алгоритмом

Основные элементы графического интерфейса:

Тип `QTextEdit`:

`guiText` - Поле для ввода текста с данными задачи

Тип `QPushButton`:

`randomButton` - Случайная генерация входных данных
`txtInputButton` - Загрузка данных из .txt файла
`guiInputButton` - Ввод данных вручную через `QTextEdit`
`toExperimentButton` - Переход к запуску алгоритма

Тип `QLineEdit`:

`txtLineEdit` - Поле ввода пути к файлу

Тип `QLabel`:

`statusInputMenu` - Статусный текст, выводящий ошибки/успехи

Графический интерфейс взаимодействует с алгоритмом с помощью сигналов

```
signals:
    void switchToExperiment(); Переход к алгоритму

    void getDataFromTxt(); Загрузить из .txt

    void getDataFromGui(); Загрузить из QTextEdit

    void generateRandomDataManager(); Случайно сгенерировать
DataManager
```

Основные элементы интерфейса эксперимента:

Данный интерфейс осуществляет визуализацию, с помощью различного текстового вывода, а также вывода графика. Он также позволяет пользователю настроить параметры алгоритма

Тип: График

`QChartView* chartView` - Визуализирует прогресс (лучшая и средняя приспособленность)

Тип: Кнопка

`QPushButton* nextStepButton` - Запустить однк итерацию

`QPushButton* toEndButton` - Запустить алгоритм до конца

`QPushButton* printPopulationButton` - Сохранить текущую популяцию в файл

`QPushButton* toInputMenuButton` - Вернуться к окну ввода данных

Тип: Выпадающий список

`QComboBox* comboFitnessFunction` - Выбор функции пригодности (жёсткая или мягкая)

`QComboBox* comboSelection` - Метод отбора (турнир, рулетка)

`QComboBox* comboCrossover` - Метод скрещивания

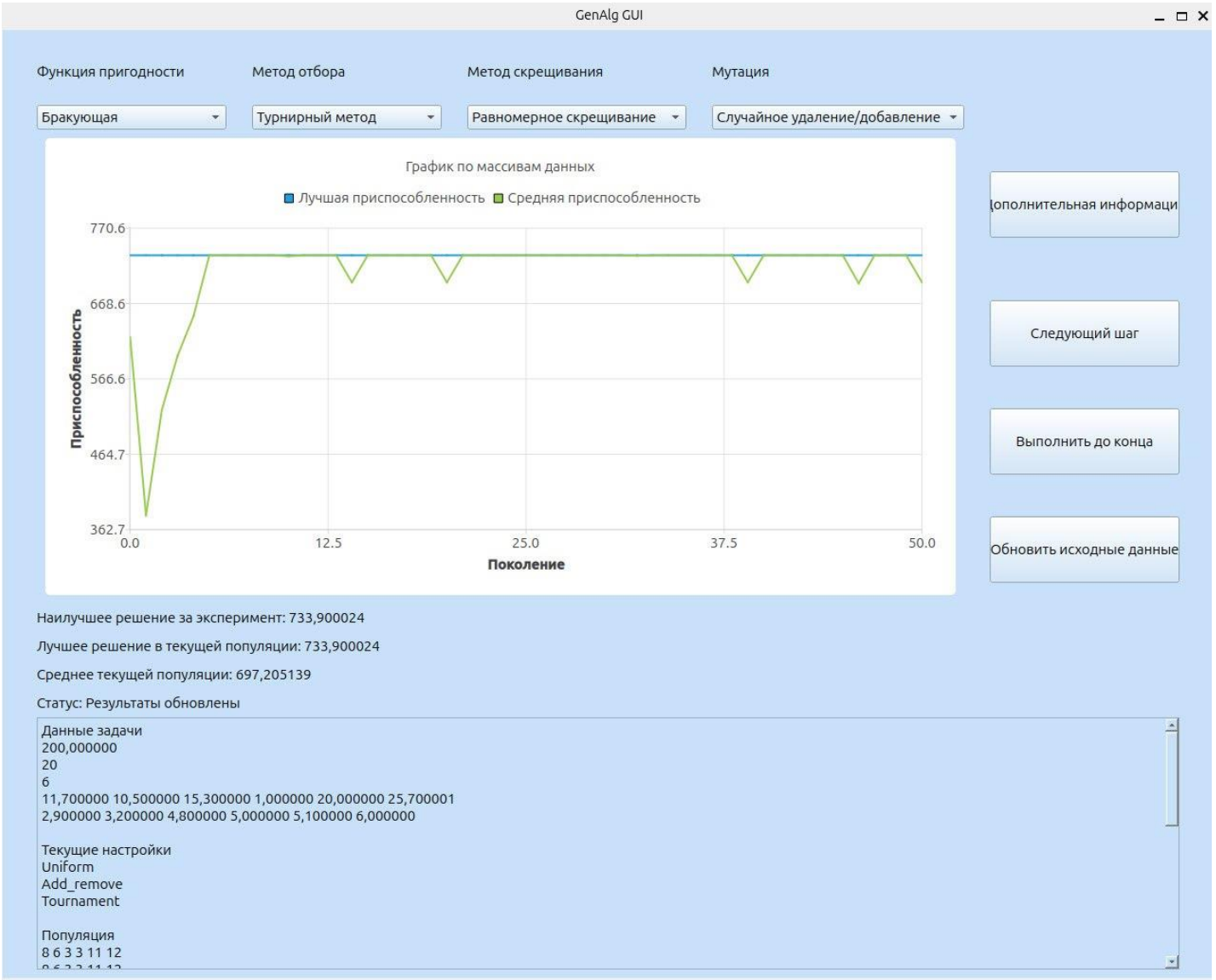
`QComboBox* comboMutation` - Мутации (удаление/добавление, обмен)

Взаимодействие с алгоритмом:

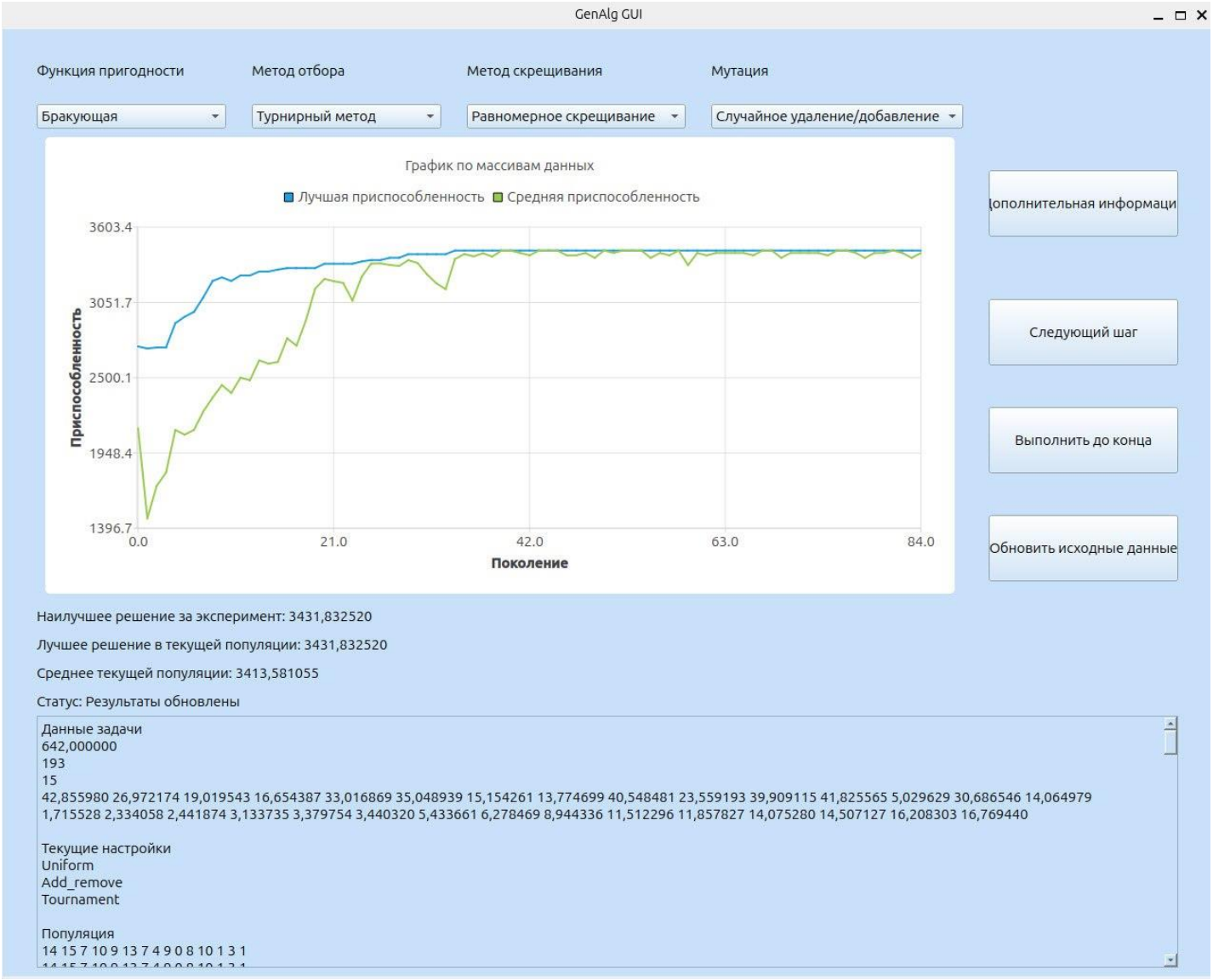
```
signals:
    void switchToInputMenu(); Вернуться к вводу данных
    void fitnessTypeSelected(int); Изменение функции пригодности
    void crossoverTypeSelected(int); Изменение скрещивания
    void mutationTypeSelected(int); Изменение мутации
    void selectionTypeSelected(int); Изменение отбора
    void runOneIteration(); Одна итерация алгоритма
    void runToTheEnd(); Алгоритм до конца
```

Примеры работы программы

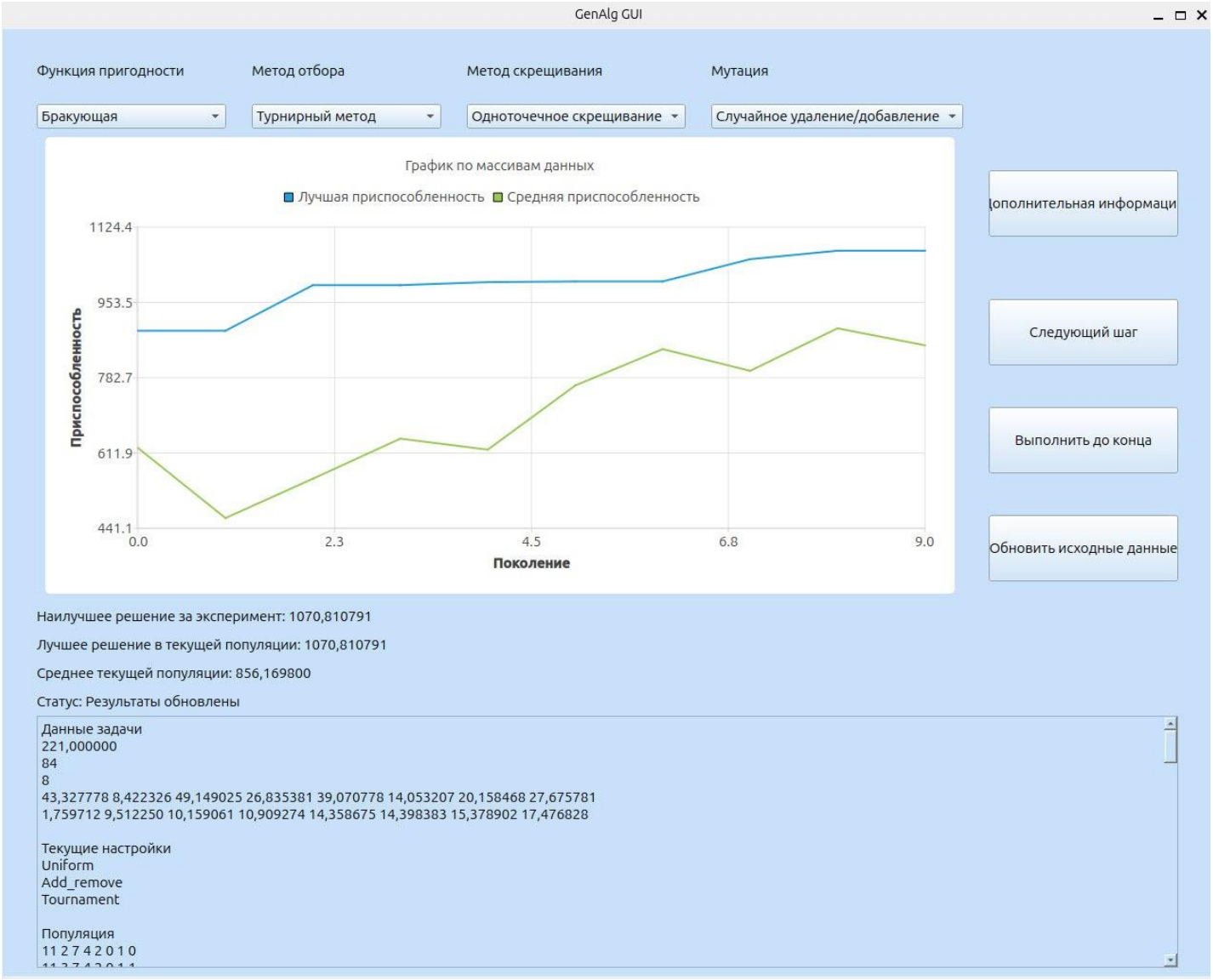
Пример 1



Пример 2


GenAlg GUI
Функция пригодности
Метод отбора
Метод скрещивания
Мутация
Бракующая
Турнирный метод
Равномерное скрещивание
Случайное удаление/добавление
График по массивам данных
Лучшая приспособленность
Средняя приспособленность
3603.4
3051.7
2500.1
1948.4
1396.7
Приспособленность
0.0
21.0
42.0
63.0
84.0
Поколение
ополнительная информаци
Следующий шаг
Выполнить до конца
Обновить исходные данные
Наилучшее решение за эксперимент: 3431,832520
Лучшее решение в текущей популяции: 3431,832520
Среднее текущей популяции: 3413,581055
Статус: Результаты обновлены
Данные задачи
642,000000
193
15
42,855980 26,972174 19,019543 16,654387 33,016869 35,048939 15,154261 13,774699 40,548481 23,559193 39,909115 41,825565 5,029629 30,686546 14,064979
1,715528 2,334058 2,441874 3,133735 3,379754 3,440320 5,433661 6,278469 8,944336 11,512296 11,857827 14,075280 14,507127 16,208303 16,769440
Текущие настройки
Uniform
Add_remove
Tournament
Популяция
14 15 7 10 9 13 7 4 9 0 8 10 1 3 1

Пример 3


GenAlg GUI
Функция пригодности
Метод отбора
Метод скрещивания
Мутация
Бракующая
Турнирный метод
Одноточечное скрещивание
Случайное удаление/добавление
График по массивам данных
Лучшая приспособленность
Средняя приспособленность
1124.4
953.5
782.7
611.9
441.1
Приспособленность
0.0
2.3
4.5
6.8
9.0
Поколение
ополнительная информаци
Следующий шаг
Выполнить до конца
Обновить исходные данные
Наилучшее решение за эксперимент: 1070,810791
Лучшее решение в текущей популяции: 1070,810791
Среднее текущей популяции: 856,169800
Статус: Результаты обновлены
Данные задачи
221,000000
84
8
43,327778 8,422326 49,149025 26,835381 39,070778 14,053207 20,158468 27,675781
1,759712 9,512250 10,159061 10,909274 14,358675 14,398383 15,378902 17,476828
Текущие настройки
Uniform
Add_remove
Tournament
Популяция
11 2 7 4 2 0 1 0