Задание

Задача коммивояжера (Traveling Salesman Problem, TSP) - это классическая задача оптимизации маршрута, которая заключается в поиске самого короткого замкнутого маршрута, проходящего через заданный набор городов, причем каждый город должен быть посещен ровно один раз.

Входные данные:

Города и их местоположение (координаты *x, y*)

Стэк используемых технологий:

random, dataclasses, math, typing, PyQt5, numpy, matplotlib

Основные понятия, которые используются в отчете:

Популяция - набор из хромосом

Хромосома - гамильтонов путь из города *'a'* в *'a'*

Ген - конкретный город из хромосомы

Настраиваемые параметры генетического алгоритма:

Размер популяции

Количество поколений

Вероятность кроссинговера (для лучше сходимости выбирают 0.8–0.95)

Вероятность мутации (для лучше сходимости выбирают 0.05–0.1)

Генерация популяции:

Стратегия "дробовика генерация достаточно большого случайного подмножества решений. В этом случае в результате эволюции есть возможность перейти в другие подобласти поиска и каждая из них имеет сравнительно небольшое пространство поиска.

Функция приспособленности (целевая функция):

Длина пути каждого набора хромосом. Вычисляется и храниться с помощью матрицы смежности.

Селекция или отбор в популяцию:

Аутбридинг на основе генотипа - первая особь выбирается случайно, а вторая наименее на нее похожая. (данный способ выбран т.к. он дает разнообразие генетического материала, повышает силу отбора). Реализация *class ExclusionSelection*

Элитная селекция - лучшие особи из текущей популяции непосредственно передаются в следующее поколение. В данном случае, только 0,1 элитных особей идут в новое поколение, остальные отбираются случайно. Это решение было принято для того, чтобы уменьшить вероятность попадания в локальный оптимум и плохие варианты тоже могли бы буть выбраны в популяцию. Реализация *class EliteSelection*

Рандомная селекция - все особи выбираются в популяцию рандомно. Очень

плохой способ селекции, был реализовон для того, чтоб посмотреть как работают другие методы в сравнении с ним. Реализация *class RandomSelection*

Выбор родителей:

Мы можем выбирать варианты в зависимости от:

фенотипа - последовательность вершин в пути

генотипа - схожесть метрик рассматриваемых хромосом

Панмиксия - случайный выбор родителей (выбран по той причине, что вероятность нахождения лучшего решения: В случае, когда лучшее решение находится в редких комбинациях генов, панмиксия увеличивает вероятность нахождения такого решения, поскольку все особи в популяции могут стать потенциальными родителями для следующего поколения). Реализация *class Panmixia*.

Метод рулетки - Метод отбора по правилу рулетки, или отбор пропорциональной приспособленности (fitness proportionate selection – FPS), заключается в случайном выборе индивидуума из популяции пропорционально его приспособленности. Предположим, у нас имеется несколько особей в популяции с разной степенью адаптации. Тогда их можно условно представить на круговой диаграмме с секторами, размером соответствующих долей. Затем, мы раскручиваем этот круг и тот сектор, на который будет указывать стрелка, будет выбран. Таким образом, отбирается особь в качестве родителя. Очевидно, что чаще (с наибольшей вероятностью) будут отбираться индивидуумы с большей приспособленностью, так как у них сектор занимает большую долю. (выбран так как сохранет особей с 'хорошими' генами и в меньше доле дает шан менее приспособленным особям). Реализация *class*

RoulleteWheelParentSelection

Турнирная селекция — сначала случайно выбирается установленное количество особей, а затем из них выбирается наиболее приспособленная особь (с лучшим значением функции приспособленности). Реализация *TournamentParentSelection*

Кроссинговер (скрещивание)

Скрещивание (также называется кроссинговер и кроссовер) базовая операция в генетическом алгоритме. Здесь перебираются пары родителей из отобранной популяции и с некоторой высокой вероятностью выполняется обмен фрагментами генетической информации для формирования хромосом двух потомков. Если родители не участвовали в скрещивании, то они переносятся (копируются) в следующее поколение.

Для задачи коммивояжера каждый потомок должен всегда содержать путь по всем вершинам, т.е. в каждой хромосоме должен содержаться набор из всех вершин и потеря этой информации считается недопутимой.

По этой причине был выбран *способ упорядоченного скрещивания* хромосом родителей. Идея его очень проста. Вначале делается двухточечное скрещивание, который не приводит к дублированию чисел.

Реализация (*class TwoPointCrossover*)

Пример:

хромосома 1 – 172|436|5

хромосома 2 – 213|765|4

->

хромосома 1 – |765|

хромосома 2 – |436|

Также можно мделать и одноточечное скрещивание, которое по сути является частным случаем двухточечного. Реализация (*class SinglePointCrossover*)

Пример:

хромосома 1 – 172|4365

хромосома 2 – 213|7654

->

хромосома 1 – |7654

хромосома 2 – |4365

А оставшиеся значения генов у потомков можно заполнить несколькими способами:

1) Мы проходим все гены первого родителя, начиная от второй точки разреза, и добавляем значения, если они не еще не присутствуют в хромосоме первого потомка. Затем, ту же самую операцию выполняем со вторым родителем и вторым потомком. В результате, получим следующий набор генов у двух потомков:

Пример:

хромосома 1 – 243|765|1

хромосома 2 – 175|436|2

2) Мы проходим все гены первого родителя, начиная с первого гена добавляем значения, если они не еще не присутствуют в хромосоме первого потомка. Затем, ту же самую операцию выполняем со вторым родителем и вторым потомком. В результате, получим следующий набор генов у двух потомков, рассматриваем одноточечное скрещивание:

Пример:

хромосома 1 – 123|7654

хромосома 2 – 217|4365

Также был примен *равномерный кроссовер*, когда при равномерном скрещивании каждый ген в потомке выбирается случайным образом от одного из родителей, если в потомке не найдены все гены, то в него записываются вершины из списка *not seen*. Реализация (*class UniformCrossover*)

Мутация

Оператор, который применяется для формирования нового поколения популяции – это мутация. Обычно, она применяется с некоторой малой вероятностью к отдельным генам потомков, меняя их определенным образом. Мутация позволяет поддерживать генетическое разнообразие особей, чтобы популяция не выродилась и хромосомы не стали похожи друг на друга.

Мутация обменом

Мутация путем обмена случайно выбранных генов. Реализация *class SwapMutation*:

1**2**35**6**47 -> 1**6**35**2**47

Мутация обращением

Здесь мы выбираем также случайным образом непрерывную последовательность генов, которые, затем, записываем в обратном порядке:

12**345**67 -> 12**543**67

(или можно использовать частный случай и переписывать гены в случайном порядке).

Реализация *class ScrambleMutation*

12**345**67 -> 12**435**67

Равномерная мутация

При равномерной мутации каждая позиция в особи имеет небольшую вероятность (0,4) быть измененной. Реализация *class UniformMutation*

Архитектура проекта

Для переключения между конкретыми операторами генетического алгоритма было принято решение использовать паттерн проектирования *Strategy*, поскольку это позволит зафикировать конкретные операторы при работе с алгоритмом, тем самым избежав случаев, когда несколько операторов одного типа работают в одном алгоритме одновременно. Каждый оператор наследуется от интерфейса типа, к которому он принадлежит. Класс, который обеспечивает выбор конкретного оператора, а так же их переключение, называется *OperatorContext*.

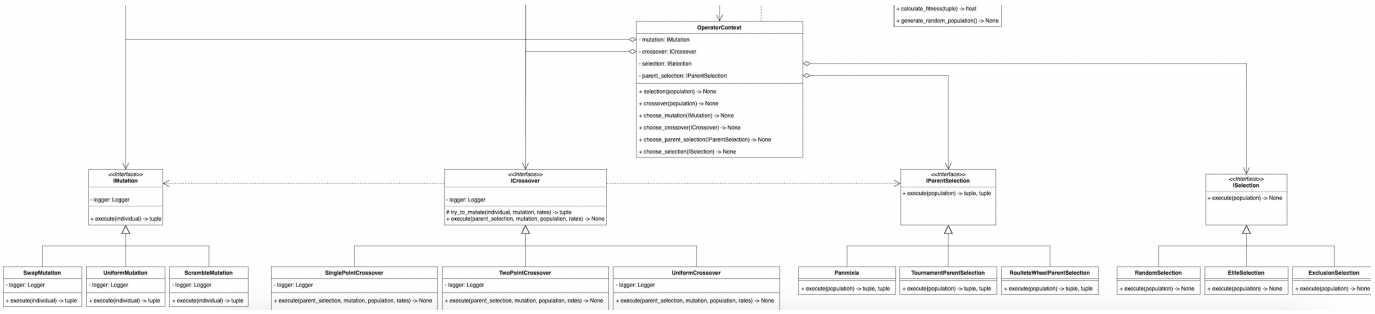

Рисунок 1 – UML-диаграмма Стретегии

Для хранения важных данных было принято решение реализовать классы: *Population*, *Caretaker*, *Logger*, а также датакласс *Rates*, хранящий вероятности мутации и скрещивания.

1. *Population* – Класс, хранящий всю информацию о некоторой популяции. Для удобства, было принято решение хранить конкретную популяцию в виде словаря, где каждый ключ является особью (картежем целых чисел, отображающий гамильтонов цикл), а значение – приспособленность особи (длина гамильтонова цикла). Также данный класс хранит матрицу смежности графа, что позволяет быстро находить длинну гамильтонова цикла у некоторой особи, а так же границы алгоритма, такие как максимальный размер популяции и число поколений.

2. $Caretaker$ – Данный класс сохраняет каждое поколение популяции и способен обрабатывать всю историю. Он нужен для последующей обработки всех итерациий алгоритма графическим интерфейсом.

3. $Logger$ – Данный класс логгирует каждый период мутации и скрещивания и хранит всю информацию в некотором списке. Также он способен занести всю информацию в некоторый источник.

Во главе иерархии реализации алгоритма стоит сам класс, отображающий его работу – ***TSP***. Он запускает алгоритм и возвращает список всех популяций полученных на каждой итерации.

Также на вершине иерархии всего проекта стоит класс $Mediator$. Он, запускает работу класса GUI, который сообщает гиперпараметры, заданные пользователем, а также информацию о вершинах, которые с помощью специального класса $Adapter$, обрабатываются в матрицу смежности, и передается на вход, совместно с гиперпараметрами, самому классу TSP. Он, в свою очередь, обрабатывает эти данные, по описаному выше принципу, после чего подает их в GUI, который отрисовывает все итерации.

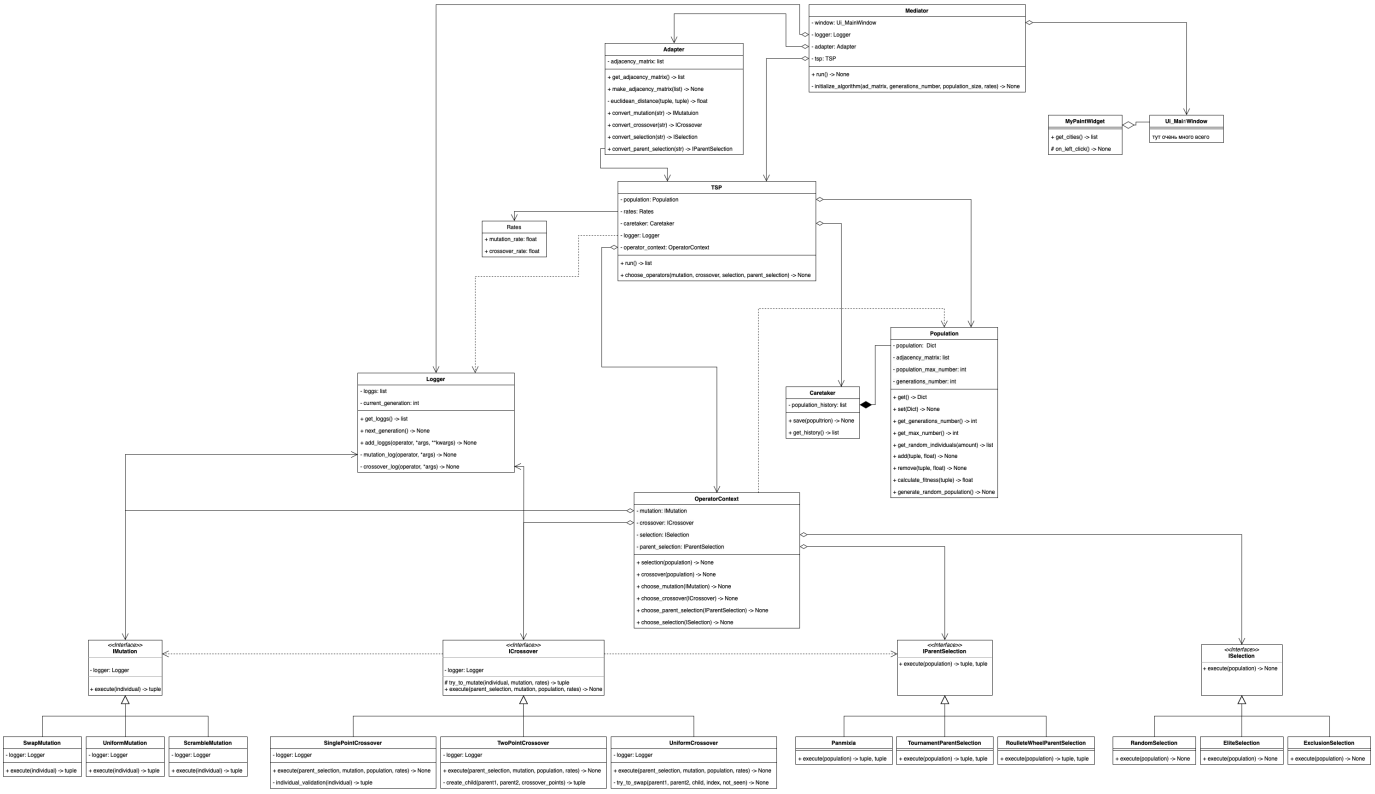

Рисунок 2 – UML-диаграмма проекта

GUI

Общий вид GUI:

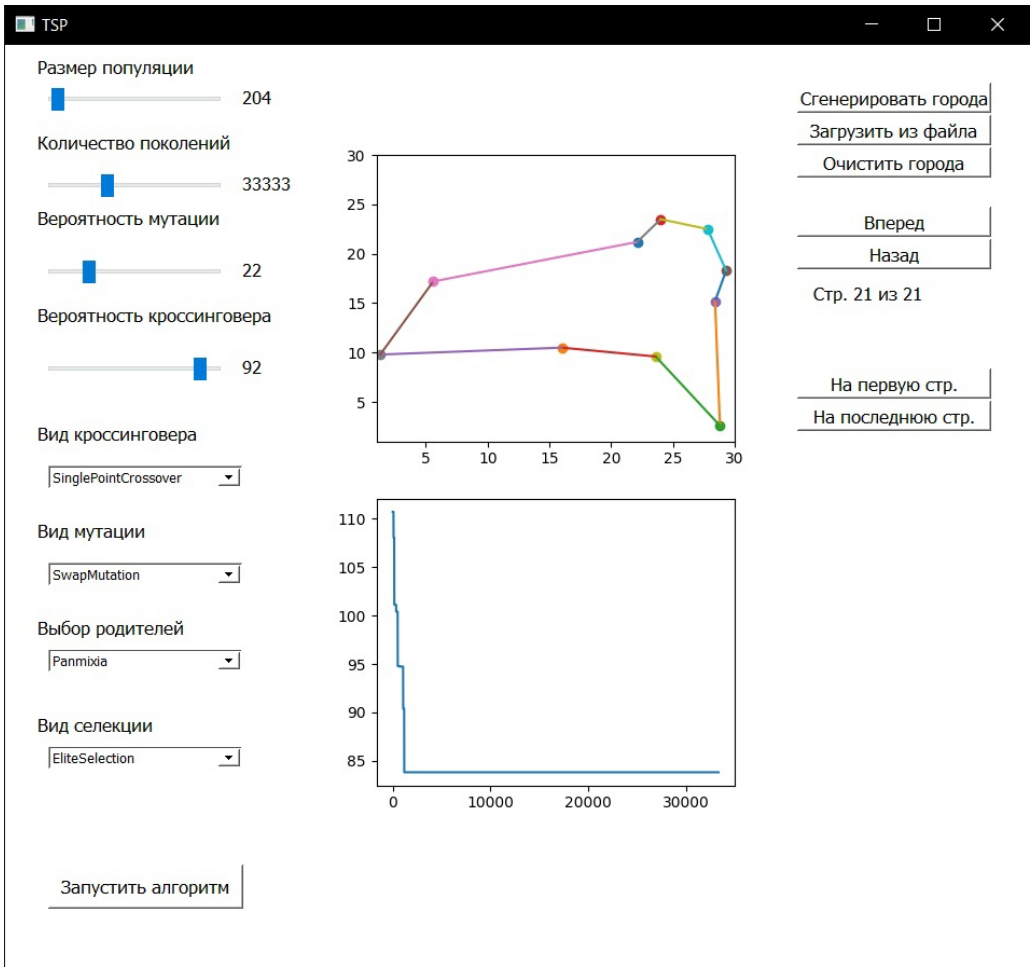

Рисунок 3 – Общий вид GUI

Для начала посмотрим на то, как выглядит GUI и ее возможности:

- слайдеры (ползунки) для выбора параметров, рядом с каждым из них есть обновляющаяся строка, которая показывает, какое значение выбрано на данный момент.

- выпадающие меню для выбора вида мутации, селекции, кроссинговера и выбора родителей

- графики для рисунков

- кнопки для случайной генерации городов, загрузки городов из файла и очистки городов

- кнопки «Вперед», «Назад», «На первую страницу», «На последнюю страницу» заблокированы до отработки алгоритма, после отработки алгоритма позволяют листать "страницы пошагово смотреть, как менялся путь коммивояжера, а также перейти прямиком на первую страницу и на последнюю страницу для сравнения пути в начале и в конце.

- кнопка «Запустить алгоритм» запускает алгоритм с выбранными на данный момент характеристиками и городами.

Для создания GUI использовалась библиотека PyQt5, для отрисовки графиков - блиблиотека matplotlib.

Графический интерфейс приложения создается как объект класса QApplication, форма - как объект класса Ui_MainWindow , который, в свою очередь, агрегирует объект класса QMainWindow библиотеки PyQt5.

Класс формы Ui_MainWindow содержит: метод setupUi, который реализует загрузку всех виджетов (включая кнопки, слайдеры, лэйблы, графики и выпадающие меню для выбора параметров); метод retranslateUi, отвечающий за первоначальную настройку виджетов; метод add_functions, отвечающий за соединение сигналов и функций, которые должны выполняться при появлении конкретного сигнала; метод start, выполняющийся при запуске алгоритма; метод set_mediator, который будет вызываться в классе Mediator для соединения GUI с медиатором и через него непосредственно с алгоритмом, и методы отрисовки.

Для отрисовки графиков создан класс MyPaintWidget, который реализован с помощью библиотеки matplotlib, он содержит два подграфика, первый - для отоб-

ражения городов и пути коммивояжера между городами, второй - для отображения графика функции приспособленности. Также у класса MyPaintWidget существует поле cities, которое инициализируется пустым списком при вызове конструктора, впоследствии туда будут записываться координаты добавленных городов в виде пар (x, y). Далее объект класса MyPaintWidgdet размещается в основной форме и используется при всех отрисовках. Поскольку объект MyPaintWidget создается при настройке формы и является полем класса Ui_MainWindow, то класс Ui_MainWindow композирует объект класса MyPaintWidget.

Реализовано три способа добавлять города на "карту":

1. Добавить город собственноручно, выбрав место левой кнопкой мыши
2. Загрузить города из файла
3. Сгенерировать города случайным образом

Скажем подробнее о каждом из способ и том, как он реализован:

1. Добавить город при помощи нажатия левой кнопки мыши можно, так как в классе MyPaintWidget сигнал нажатия левой кнопки мыши подключен к методу _on_left_click, который, в свою очередь, добавляет координаты города в список cities, и отрисовывает город в соответствии с координатами.

2. Загрузка городов из файла происходит следующим образом: при нажатии на кнопку "Загрузить из файла открывается диалоговое окно, в котором пользователь может выбрать файл, который следует открыть. Диалоговое окно реализовано с использованием QFileDialog, окно открывается в директории проекта. Реализована проверка корректности координат, находящихся в файле - города с некорректными координатами игнорируются. Чтение производится при помощи класса Reader: путь

до файла, полученный при помощи QFileDialog, передается в функцию read_data класса Mediator, в котором создается объект класса Reader и вызывается его метод read_data.

3. Генерация городов производится при помощи класса Generator, связь с ним аналогична связи с классом Reader: вызывается метод generate_data класса Mediator, в котором создается объект класса Generator и вызывается его метод generate_data.

Для случаев 2 и 3 данные о городах, полученные в медиаторе, затем вносятся в общие данные о городах при помощи метода set_cities класса MyPaintWidget (в данном методе список городов просто расширяется списком городов, полученным в качестве параметра). Это делается для того, чтобы можно было мгновенно отрисовать новые города, а также использовать при вводе городов комбинацию из нескольких способов 1-3 в любом порядке.

Также стоит отметить, что любой из способов 1-3 можно использовать сколь угодно много раз.

Получение данных из алгоритма происходит также при помощи класса Mediator. В методе run класса Mediator ининциализируется и запускается алгоритм, после чего история популяций передается в GUI для отрисовки. Mediator агрегирует объект класса Ui_MainWindow, именно с помощью этой связи результат алгоритма передается в GUI.

Примеры выполнения и обзор результатов

Для начала приведем простой тест из 6 городов, расположенных случайным образом и посмотрим на результат работы. Для столь малого количества городов можно выбрать низкие значения размера популяции и количества поколений. Выбранные метрики можно увидеть на скрине. Как видим, график функции приспособленности

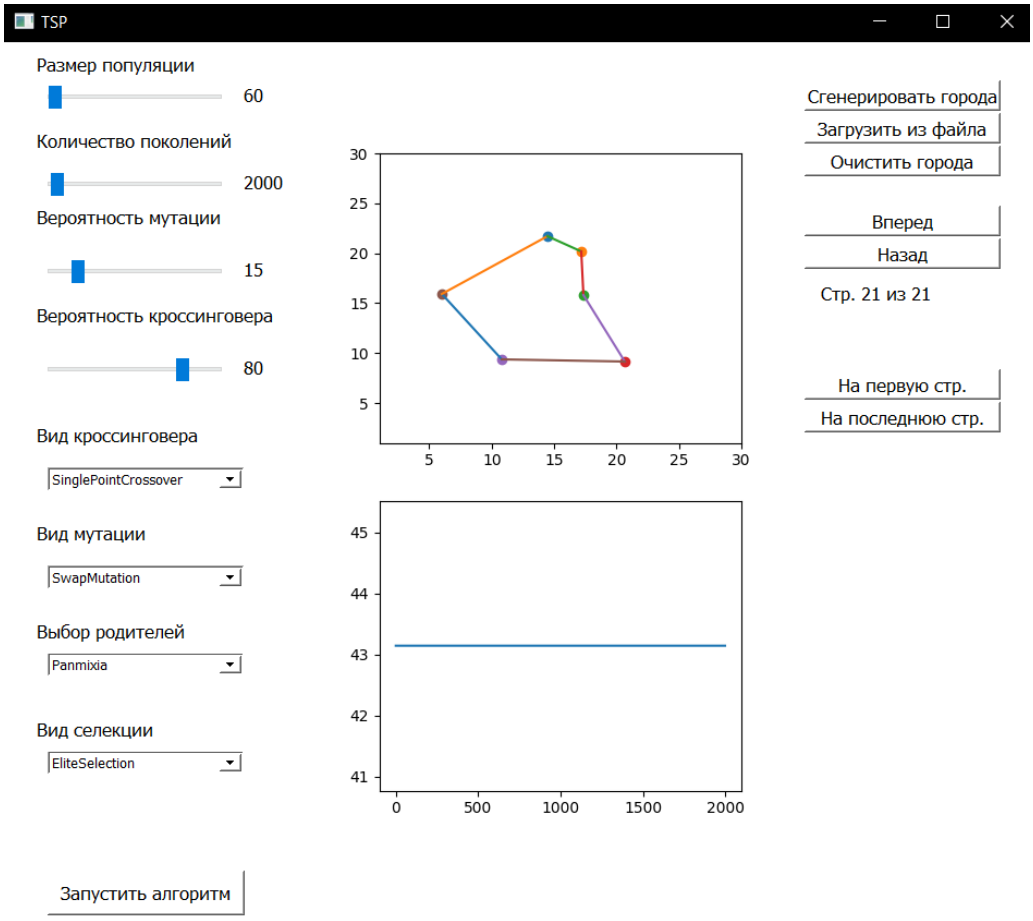

Рисунок 4 – Базовый пример с 6 городами

выглядит как прямая, а это означает что наилучший путь уже существовал среди первого поколения. В данном примере график путь коммивояжера не меняется, если листать от последней страницы к первой.

Приведем более интересный пример: сгенерируем 15 городов случайным образом и попробуем найти путь коммивояжера для них. В данном случае имеет смысл

увеличить число поколений, чтобы с большей вероятностью найти оптимальный путь.

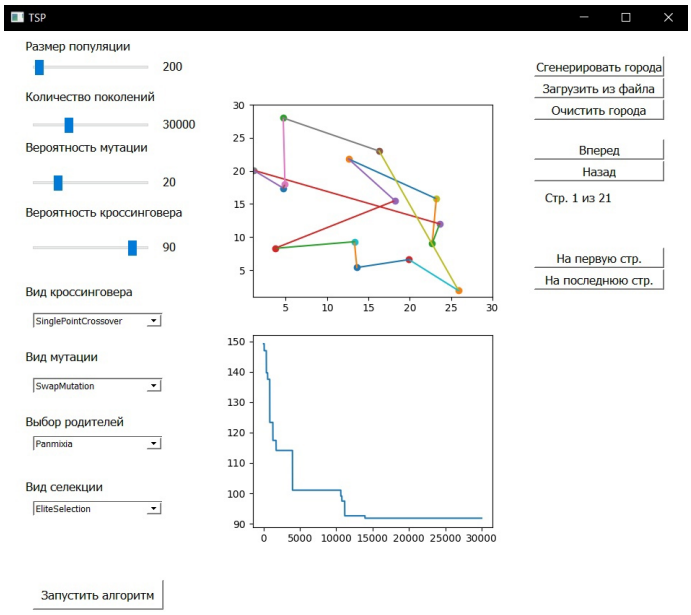

Рисунок 5 – Пример с 15 случайно расположенными городами, первая страница

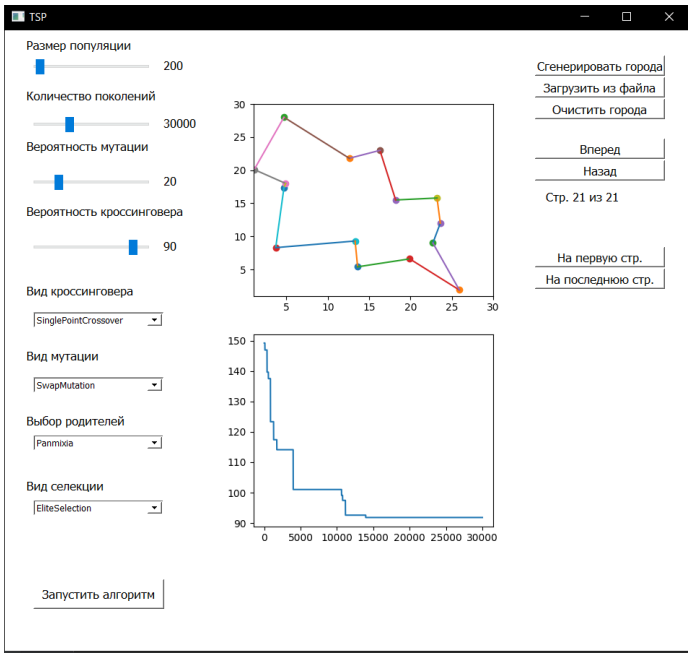

Рисунок 6 – Пример с 15 случайно расположенными городами, последняя страница

Можем увидеть, как улучшился путь. Учитывая, что в контексте задачи су-

ществует евклидово пространство, оптимальность пути можно оценить визуально. Также видим, что в данном случае функция приспособленности - это невозрастающая ступенчатая функция.

Для предоставления корректности работы алгоритма введем еше один пример: расположим 20 городов по окружности. Очевидно, что при условии евклидова пространства кратчайший гамильтонов цикл будет проходить подряд, а график путь коммивояжера должен быть приближен к окружности. Видим, что , если на первой

Рисунок 7 – Пример с 20 городами, расположенными по окружности, первая страница

странице путь коммивояжера в большинстве ребер шел по хордам, то на последней странице путь коммивояжера идет строго по окружности, чего мы и хотели добиться.

Благодаря этому и предыдущим примерам корректность работы алгоритма доказана.

Рисунок 8 – Пример с 20 городами, расположенными по окружности, последняя страница

Список литературы

- Методичка ГА
- Статья по ГА
- Статья на википедии
- Статья на НОУ ИНТУИТ