**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**

Федеральное государственное бюджетное образовательное учреждение высшего образования

**«Сибирский государственный университет науки и технологий  
имени академика М.Ф. Решетнева»**

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники

**КУРСОВАЯ РАБОТА**

по дисциплине «Алгоритмы и структуры данных»

на тему: «Программная реализация решения задачи коммивояжера с помощью генетического алгоритма»

                                                             Выполнил: студент группы БПИ18-01

                                                             очной формы обучения

*Рылов Артем Евгеньевич*

               Руководитель:

                                                             канд. техн. наук, доцент В.В. Тынченко

Дата сдачи: «\_\_\_\_\_»\_\_\_\_\_\_\_\_20\_\_\_г.

Дата защиты: «\_\_\_\_»\_\_\_\_\_\_\_\_\_20\_\_г.

Оценка:\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_             \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

                                                                                                                                                 (подпись руководителя)

Красноярск 2018

Институт информатики и телекоммуникаций

Кафедра информатики и вычислительной техники



|  |  |  |  |
| --- | --- | --- | --- |
| **ЗАДАНИЕ**  на курсовую работу по дисциплине «Алгоритмы и структуры данных»  студенту Рылову Артему Евгеньевичу | | | |
| Группа БПИ18-01 |  | Форма обучения очная |  |
| 1. Тема работы: Программная реализация решения задачи коммивояжера с помощью генетического алгоритма | | | |
| 2. Срок сдачи студентом работы: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | |
| 3. Перечень вопросов, подлежащих разработке при написании теоретической части:  1. Описание точных алгоритмов  2. Описание эвристических алгоритмов  3. Подробное описание генетического алгоритма | | | |
| 4. Перечень вопросов, подлежащих разработке при написании практической части:  1. Выбрать язык программирования для реализации данного алгоритма  2. Системные требования для запуска программы  3. Описание программы  4. Порядок эксплуатации программы  5. Сравнить эффективность генетического алгоритма | | | |
| 5. Дата выдачи задания: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | |
| Руководитель: Тынченко В.В.    \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | |
| (Подпись) | | | |
|  | | | |
| Задание принял к исполнению (дата): \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | | | |
| (Подпись студента) | | | |

**СОДЕРЖАНИЕ**

Введение4

1 Анализ области решения задачи коммивояжера6

1.1 Точные алгоритмы6

1.2 Эвристические алгоритмы7

2 Описание генетического алгоритма10

2.1 Общее описание генетического алгоритма10

2.2 Генетический алгоритм в решении задачи коммивояжера11

3 Программная реализация12

3.1 Выбор языка программирования12

3.2 Системные требования для запуска программы12

3.3 Описание программы13

3.4 Порядок эксплуатации программы15

4 Сравнение эффективности генетического алгоритма18

Заключение21

Список использованных источников22

Приложения А-Г23-29

**ВВЕДЕНИЕ**

*Задача коммивояжера* (Travelling salesman problem, сокращенно - TPS), ну или *задача путешественника-торговца(странствующего торговца)* – это одна из самых известных задач из раздела комбинаторной оптимизации, суть ее состоит в том, что нужно найти самый выгодный маршрут, который будет проходить через указанные города по одному разу и затем возвращаться в исходный город.

Эта задача появилась очень давно и до сих пор актуальна.

В 18 веке, в 19 веке и в настоящее время у людей возникают ситуации, в которых им нужно пройти по всем определенным пунктам, при этом затратить как можно меньше времени: это может быть почтальон, который собирается разносить письма и посылки, в пределах улицы, города, не важно; это может быть водитель грузовика, который развозит некий продукт по нескольким местам подряд; также, это может быть грузовое судно, которому нужно развезти несколько тысяч тонн груза по нескольким странам/портам/островам.

Во всех этих случаях от того, насколько *короткая и выгодная* будет выбрана дорога, будет зависеть количество денег, потраченное на транспортировку данных грузов; количество горючего, которое поглотила техника, с помощью которой доставляли груз; время, потраченное на путь, которое тоже оплачивается и несет убытки транспортной компании; конечно же, во всех случаях, кроме первого, там фигурирует лишь время почтальона.

Как раз-таки, из-за подобных случаев и возникла *задача коммивояжера*: люди хотели сохранить свои деньги, топливо, силы и т. д. там, где это возможно и даже нужно.

Первая работа, в которой была описана данная проблема, была написана в 1832 году, название ее было – «Коммивояжер – как он должен вести себя и что должен делать для того, чтобы доставлять товар и иметь успех в своих делах – советы старого курьера». К сожалению, математического аппарата для решения *задачи коммивояжера,* там не было, но в ней были предложены некоторые примеры маршрутов для некоторых регионов Германии и Швейцарии.

Ранним вариантом задачи коммивояжера можно считать «Icosian Game» Уильяма Гамильтона, который был написан в 19 веке, там суть заключалась в том, что нужно найти маршруты в графе с двадцатью узлами.

А первое упоминание, именно в качестве *математической* задачи на оптимизацию, принадлежит Карлу Менгеру. Он в 1930 сформулировал ее на математическом коллоквиуме так: «Мы называем проблему посыльного (поскольку этот вопрос возникает у каждого посыльного, в частности, ее решают все путешественники) задачу найти кратчайший путь меж конечным множеством мест, расстояние меж которыми известно»

Из этой фразы и появилось, уже известное нам, название *задачи коммивояжера – задача странствующего торговца.*

Далее, эта задача изучалась учеными все глубже и глубже. Причем не с практической точки зрения, а с теоретической, в качестве модели, с помощью которой разрабатывали новые алгоритмы оптимизации.

В 1960-1970-е годы данная задача изучалась учеными из разных областей науки как теоретически, так и с точки зрения приложений в различных науках таких как: информатика, экономика, биология.

Многие современные распространенные методы *дискретной оптимизации*, по типу *метод отсечения, ветвей и границ* и другие различные варианты алгоритмов, называемых эвристическими, разработались с помощью *задачи коммивояжера.*

*Актуальность.* Данная задача активно изучается современными учеными, так как логистика перевозок, то есть, тот самый поиск самого короткого и выгодного маршрута, очень важна для всех больших транспортных компаний и не только.

*Цель*. Целью курсовой работы является программная реализация решения задачи коммивояжера с помощью генетического алгоритма.

*Задачи*. К задачам курсовой работы относятся:

1. Провести анализ известных методов решения *задачи коммивояжера.*

2. Описать методы, более интересные для нас.

3. Спроектировать и реализовать программу, которая будет решать *задачу коммивояжера* с помощью генетического алгоритма.

4. Выполнить тестирование программы.

**1 АНАЛИЗ ОБЛАСТИ РЕШЕНИЯ ЗАДАЧИ КОММИВОЯЖЕРА**

Можно выделить два типа алгоритмов, для решения задачи коммивояжера: точные алгоритмы и эвристические алгоритмы.

Отличие их состоит в том, что одни алгоритмы дают точный ответ на рассматриваемую задачу, а другие дают приближенный ответ. Опишем некоторые из них ниже.

**1.1 ТОЧНЫЕ АЛГОРИТМЫ**

[4] Особенность этих алгоритмов, как можно понять из названия, является то, что они дают гарантированно самый короткий маршрут, точное решение поставленной задачи.

Мы рассмотрим два точных алгоритма: *алгоритм полного перебора (АПП)* и *метод ветвей и границ (МВиГр).*

*Алгоритм полного перебора (АПП)* – осуществляет поиск в пространстве N! Посредством перебора всех вариантов, где N – кол-во узлов, в данной задаче коммивояжера.

Результатом работы алгоритма, при любых входных данных, будет точное решение, с самым коротким путем.

Недостатком *алгоритма полного перебора* является алгоритмическая временная сложность – пространство поиска растет экспоненциально, поэтому, когда N достаточно мало, мы можем использовать этот алгоритм, но если N не является значительно малым, то этот алгоритм не используется на практике, так как вычисления будут занимать очень много времени. В таких случаях используют эвристические и поисковые алгоритмы, о которых будет рассказано ниже.

*Метод ветвей и границ (МВиГр) –* это есть некоторое развитие *алгоритма полного перебора.* Суть этого алгоритма заключается в том, мы добавляем проверки критерия ограничивающей функции, исходящего из условия задачи, по которому, на определенном уровне, можно приостановить построение данной ветви дерева перестановок.

Этот алгоритм содержит и сохраняет в себе все положительные свойства *алгоритма полного перебора,* также решает задачу коммивояжера со 100% точностью, но все же ненамного лучше подходит для решения задач, где N – есть число не значительно малое, а большое.

К преимуществам данного алгоритма мы сможем отнести такую возможность, как возможность распараллеливания и точное решение задачи.

**1.2 ЭВРИСТИЧЕСКИЕ АЛГОРИТМЫ**

[4] *Эвристические алгоритмы –* такие алгоритмы решения задачи, которые включают практический метод, не являющийся точным или оптимальным со 100% вероятностью, но эти алгоритмы дают такой ответ, точности которого будет достаточно для решения поставленной практической задачи.

Эти алгоритмы более интересны для нашего рассмотрения, так как вообще не все задачи в логистике могут быть решены с помощью точных алгоритмов.

Эти алгоритмы позволяют ускорить решение задач в случаях, когда точное решение не может быть найдено по ряду причин.

Из *эвристических алгоритмов* будут приняты в рассмотрение пять алгоритмов: *случайный метод,* *метод включения дальнего (МВД),* *BV-метод, муравьиный алгоритм* и *генетический алгоритм.*

*Случайный метод (СМ)* – метод, суть которого заключается в том, что мы выбираем несколько случайных решений интересующей нас задачи и выбираем из них самое подходящее и оптимальное.

Сложность этого алгоритма является N!, он редко дает подходящий для практической задачи ответ и его очень редко используют для решения задачи коммивояжера.

*Метод включения дальнего (МВД) –* алгоритм, суть которого заключается в следующем: точки (города), которые максимально отдалены друг от друга, никогда не смогут быть смежными в цепи. Эти две точки (города) и будут базовыми для дальнейшего решения. После этого опять находится вершина, которая максимально удалена от вершин, уже заключенных в цепь. Затем находится минимальная сумма длин ребер между найденной вершиной и парой смежных вершин в цепи, это действие задает место вершины, которую мы нашли, в цепи.

Сложность данного алгоритма является линейной, он дает приблизительное решение задачи, то есть решение, не являющееся гарантированно самым эффективным. Также, он не может быть распараллелен.

*BV-метод –* алгоритм, основанный на анализе полученного маршрута, который мы будет считать эталонным, и его оптимизации. Алгоритм условно можно разделить на два этапа:

1) получение начального эталонного решения (пути);

2) оптимизация начального решения.

Начальное решение представляет собой лучшее решение из всех решений, которые мы получили на основе «жадного» алгоритма. «Жадные алгоритмы» - алгоритмы, заключающиеся в принятии локально оптимальных решений на каждом этапе, при этом допуская, что, конечно, решение также может быть оптимальным.

Второй этап состоит в улучшении полученного исходного эталонного решения (маршрута) с помощью *BV-модификаторов*, суть которых – это выявление неоптимальных участков и преобразование их.

Сложность данного алгоритма является квадратичная сложность, он дает приблизительное решение задачи, то есть решение, не являющееся гарантированно самым эффективным. Этот алгоритм может быть распараллелен, но лишь на втором этапе.

*Генетический алгоритм (ГА)* – алгоритм, который дает ответ на задачу оптимизации и моделирования с помощью случайного подбора, вариации и комбинирования искомых параметров, также используя механизмы, которые аналогичны естественному отбору в природе.

Этот алгоритм использует методы естественной эволюции: мутации, наследование, кроссинговер и отбор, что и делает его разновидностью *эволюционных вычислений*, которые применяются при оптимизации. Но *генетический алгоритм* отличается от остальных тем, что он использует метод (оператор) «скрещивания», с помощью которого производятся операции рекомбинации решений-кандидатов, роль «скрещивания» такая же, как и роль у скрещивания в живой природе.

Имеет недостаток в виде преждевременной сходимости, то есть не всегда может выйти из локального минимума.

Дает приближенное решение задачи, может быть распараллелен.

*Муравьиный алгоритм (МА)* – алгоритм, который использует для решения задачи подход, заключающийся в применении и анализе модели поведения *муравьев*, которые ищут маршруты (пути) от своей колонии к пище (источнику питания).

Как уже говорилось, алгоритм основан на поведении *муравьиной колонии*, то есть, на маркировки более удачных маршрутов (путей) большим количеством феромона (продукты внешней секреции, которые выделяются муравьями, и не только). Алгоритм начинается с «разбрасывания» муравьев в городах (вершинах графа), после этого начинается движение муравьев, а направление движения определяется вероятностным методом, с помощью формулы вида:

Где – вероятность перехода по пути i,

– величина, обратная весу (длине) i-го перехода,

– количество феромона на i-ом переходе,

q - величина, определяющая «жадность» алгоритма,

p - величина, определяющая «стадность» алгоритма,

Так же, на основе *муравьиного алгоритма* был развит *алгоритм «Системы муравьев»*. Вот в чем отличия:

- в выборе нового города явно задается баланс использованием предыдущих знаний и нахождением новых решений;

- при глобальном обновлении феромона, то есть по завершению каждой итерации, добавление феромона происходит только по дугам, которые принадлежат глобальному лучшему пути;

- в то время, пока муравьи заняты поиском решения, будет происходить локальное обновление феромона, то есть полная аналогия с настоящими муравьями;

При программной реализации решения *алгоритма коммивояжера* в данной курсовой работе будет использоваться *генетический алгоритм,* так как он является одним из самых эффективных при решении данной задачи и более интересен с точки зрения изучения.

**2 ОПИСАНИЕ ГЕНЕТИЧЕСКОГО АЛГОРИТМА**

**2.1 ОБЩЕЕ ОПИСАНИЕ**

[1, 2, 3] Принято считать, что отцом *генетических алгоритмов* является ученый теоретик *Джон Холланд (John Holland)*, написавший в 1975 году книгу «Adaptation in Natural and Artificial System» издательства Мичиганского университета. Для справки, так же в 1975 (в том же) году появился первый персональный компьютер, именуемый «Альтаир 8800». Но работа Джона Холланда своими корнями уходит к биологам 1950-х годов, которые пытались моделировать эволюцию на компьютерах.

И с тех пор, *генетический алгоритм*, как и другие алгоритмы оптимизации, использовался для решения самых разных задач, таких как:

1. Нахождение формы концертного зала с оптимальными акустическими характеристиками.

2. Создание оптимальной формы крыла для сверхзвукового самолета.

3. Составление библиотеки химических веществ, для синтеза различных лекарств.

4. Проектирование различных микросхем, которые подходят для решения определенной задачи.

И еще очень большого спектра задач.

*Принцип работы алгоритма* взят у естественной природы, он полностью строится на концепции естественного отбора. С помощью *перебора* и *отбора* получается оптимальная комбинация генов для рассматриваемой задачи.

Алгоритм делится на три этапа:

1. Селекция

2. Скрещивание, мутация

3. Формирование нового поколения

Обычно, алгоритм завершает свою работы после заданного количества итераций (количества поколений).

*Подробней о шагах*:

*- создание начальной популяции*. На этом шаге создается начальная популяция, из которой, вполне возможно, не будет ни одного хорошего решения, но очень велика вероятность, что алгоритм эту проблему исправит и сформирует много подходящих решений на следующих шагах. Самое главное на этом шаге – решения (особи) должны быть одного «формата», вида и должны быть способны к скрещиванию;

*- размножение (скрещивание)*. Размножение делается таким образом, чтобы потомок получил черты от всех своих родителей, их количество может быть разным: два, три и, в некоторых случаях, даже один. Причем, размножаться могут не только выжившие особи, но и не выигравшие естественный отбор, в этом, конечно, есть различие с живой природой;

*- мутации.* Мутации устроены таким образом, что мы незначительно изменяем одну из особей. Мутировать так же могут особи, которые не прошли естественный отбор. Вероятность мутации задается в качестве параметра;

*- отбор.* Мы отбираем из популяции те особи, которые больше всех удовлетворяют поставленной задачи. При этом долю выживших мы определяем заранее, указывая в виде параметра.

**2.2 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ В РЕШЕНИИ ЗАДАЧИ КОММИВОЯЖЕРА**

[5] Теперь о том, как использовать *генетический алгоритм* при решении *задачи коммивояжера.*

*Геном*, в нашем случае, будет являться номер точки (города), через который проходит маршрут коммивояжера.

*Хромосома (особь) –* последовательность *генов*, которые определяют замкнутый маршрут, проходящий через все заданные точки (города), причем по одному разу через каждую точку.

*Популяция* – набор из маршрутов, которые подходят через все точки по одному разу, то есть, удовлетворяют условию задачи.

*Мутация* будет проходить по такому принципу: мы будем менять местами у мутирующей особи 2 гена, расположенных по соседству.

*Скрещивание,* в случае *задачи коммивояжера*, очень трудно реализовать при двух родителях, так как получившийся ген может не соответствовать условиям задачи. Например, 04325160 и 062435710, при их скрещивании может получиться 043245710, в результате получается решение, которое не подходит под критерии задачи. Намного эффективнее и проще реализовать скрещивание с одним родителем, как бы это странно ни звучало. Мы будем разворачивать ген на 180 градусов между двумя случайными точками внутри него. Например, из 052|346|710 получается 052|643|710.

*Отбор* будет осуществляться по такому принципу: особи, которые показывают лучшее время, переходят в новое поколение.

Теперь, когда понятно как реализовывать *генетический алгоритм* при решении *задачи коммивояжера*, можно переходить к программной реализации.

**3 ПРОГРАММНАЯ РЕАЛИЗАЦИЯ**

**3.1 ВЫБОР ЯЗЫКА ПРОГРАММИРОВАНИЯ**

Генетический алгоритм можно без особых проблем реализовать на многих популярных языках программирования. Рассмотрим претендентов.

1. *C*

2. *Python*

3*. C++*

4*. Delphi*

5*. Go*

6*. Java*

Можно реализовать данный алгоритм на всех вышенаписанных языках программирования, так как он не требует каких-то особых библиотек, которые распространены лишь на одном из этих языков.

Но выбор остановился на *Python*, так как это самый лаконичный и читабельный язык из всех вышеперечисленных. Да и к тому же, на нем проще реализовывать практически все алгоритмы, так как он является *динамически типизируемым* языком.

**3.2 СИСТЕМНЫЕ ТРЕБОВАНИЯ ДЛЯ ЗАПУСКА ПРОГРАММЫ**

Программа написана на python 3.7, поэтому для ее запуска необходим установленный интерпретатор python 3.7 или более поздней версий. В ней используются только стандартные библиотеки, такие как *typing, sys, math, re, time,* поэтому для ее запуска не нужно устанавливать никакие модули.

**3.3 ОПИСАНИЕ ПРОГРАММЫ**

Было реализовано 2 алгоритма оптимизации:

1. *Случайный алгоритм* оптимизации. Этот алгоритм был реализован для сравнения с *генетическим алгоритмом*, чтобы понять, насколько хорошо работает второй.

2*. Генетический алгоритм*. Основной алгоритм оптимизации в этой работе, разработка которого и есть цель курсовой работы.

Сначала опишем, что принимает и возвращает функция *случайной оптимизации*, которая размещена в файле random\_optimization.py. Принимает она 4 параметра: points – список из всех точек, по которым нужно пройтись, cost\_func – функция, рассчитывающая стоимость маршрута, max\_iter – количество итераций алгоритма, log – если True, то будет происходить документирование. Возвращает она 2 значения: best\_length – длину лучшего маршрута, best\_path – лучший путь.

Теперь опишем реализацию *случайного алгоритма* построчно (см. приложение А):

**1-2**: подключение необходимых модулей

**5-9**: сигнатура функции случайной оптимизации

**10-15:** комментарии к функции

**17-21:** если включен режим документирования, подготавливаемся для вывода прогресса выполнения алгоритма

**23-24:** задаем значения переменный лучшего длинны маршрута и лучшего маршрута максимально большие значения, для дальнейшего сравнивания

**26:** входим в основной цикл *случайного алгоритма*

**27-29:** если включен режим документирования, выводим процесс выполнения алгоритма (в процентах кратных 10)

**33:** формирование списка из номеров точек, по которым нужно пройти

**34-35:** выбор первой точки (всегда первая) и удаление ее из списка

**38-43:** выбор случайного решения

**46:** высчитывание длины полученного случайного решения

**47-49:** сравнивание длины полученного случайного решения с лучшей полученной длиной и если первое больше, то присваиваем его лучшей длине и текущий маршрут лучшему маршрут

**50:** выход из основного цикла алгоритма

**51-53:** если включен режим документирования, выводим процесс выполнения алгоритма

**55:** возвращаем лучшую длину и лучший маршрут

Перейдем к *генетическому алгоритму*. Опишем, что принимает и возвращает функция *генетической оптимизации*, которая расположена в файле genetic\_optimization.py*.* Принимает она 7 параметров: domain – список из всех генов, cost\_func – функция, рассчитывающая стоимость особи, pop\_size – размер популяции, mut\_prob – шанс мутации, elite – процент выигравших отбор особей, max\_iter – количество итераций алгоритма, log – если True, то будет происходить документация.

Сейчас опишем реализацию данного алгоритма построчно (см. приложение Б):

**1-2:** подключение необходимых модулей

**5-12:** сигнатура функции *генетической оптимизации*

**13-22:** комментарии к функции

**24-25:** если включен режим документирования, подготавливаемся для вывода прогресса выполнения алгоритма

**28-30:** описание мутации особи

**33-36:** описание скрещивания особи

**39-55:** создание первой популяции из случайных особой (алгоритм подобный тому, что используется в *случайной оптимизации)*

**58:** подсвет, сколько особей перейдет в следующее поколение

**60-61:** подготовка для вывода прогресса выполнения алгоритма

**64:** вход в основной цикл алгоритма

**65-67:** если включен режим документирования, выводим процесс выполнения алгоритма (в процентах кратных 10)

**69-75**: отсеивание особей, не прошедших естественный отбор

**78-86:** добавление особей, полученных мутацией и скрещиванием до начального размера популяции

**88-89:** составляем список со стоимостью особей и самими особями и сортируем его

**91-93:** если включен режим документирования, выводим процесс выполнения алгоритма

**96:** возвращаем лучшую стоимость особи и лучшую особь

Были написаны функции для определения расстояния между точками, высчитывания общей длины маршрута, загрузки точек из файла, замера длительности выполнения алгоритма и показа результата алгоритма. Все эти функции определены в файле functions.py. Опишем этот файл построчно (см. приложение В):

**1-4:** подключение необходимых модулей

**7-14:** определение функции для определения расстояния между 2 точками. Данная функция принимает в качестве параметров 2 точки и возвращает расстояние между ними.

**15-21:** определение функции для расчета общей длины маршрута. Она принимает в качестве параметров список всех точек и список из номеров точек (маршрут) и возвращает общую длину маршрута

**24-37:** определение функции для загрузки точек из файла. В качестве параметров принимает имя файла, из которого нужно загрузить точки. Точки в файле должны быть в таком формате: (x, y). Файл может содержать сколько угодно точек. Возвращает список точек.

**40-51:** определение функции для замера времени, потраченного на алгоритм. Функция принимает в качестве параметра список всех точек, функцию для расчета стоимости маршрута, оптимизационную функцию, количество итерация для оптимизационную функции, и True или False, в зависимости от того, хотим ли мы документировать процесс выполнения функции. Возвращает длину маршрута, маршрут, полученные с помощью оптимизационной функции, и время, затраченное на выполнение алгоритма.

**54-63:** определение функции для вывода результата работы функции. Принимает 4 параметра: имя функции, длину маршрута, маршрут, время, затраченное на алгоритм. Ничего не возвращает, но выводит на экран всю информацию об результатах алгоритма.

У разработанной программы есть 2 режима работы: одноразовый запуск *генетического* и *случайного алгоритмов* и вывод их результатов и режим накопления статистики, который несколько раз выполняет алгоритм и записывает в файл (или выводит в консоль) полученную длину маршрута и время, затраченное на поиск этого маршрута. Оба этих режима описаны в виде функций в файле modes.py, опишем построчно (см. приложение Г):

**1-6:** подключение необходимых модулей

**9-25:** определение функции, отвечающей за одноразовый запуск алгоритмов. Функция принимает в качестве параметров список всех точек, по которым нужно пройти и количество итераций для *генетического* и *случайного алгоритма.* Она ничего не возвращает, но выводит на экран результаты работ обоих алгоритмов.

**28-47:** определение функции, отвечающей за накопление статистики. Принимает 7 параметров: список точек, по которым нужно пройтись, функцию оптимизации, имя функции оптимизации, минимальное количество итераций, максимальное количество итераций, количество шагов и поток, куда производить запись статистики. Ничего не возвращает, записывает в файл (или выводит на экран) статистику по найденной длине маршрута и времени, затраченного на поиск этой длины при нескольких выполнениях алгоритма.

И последний файл, в котором описано пользовательское меню, это файл main.py, опишем его построчно (см. приложение Д):

**1-6:** подключение всех необходимых модулей

**10:** загрузка всех точек, по которым нужно пройтись, из файла points

**12:** взятие у пользователя номера режима

**15-20:** если пользователь выбрал 1 режим, то берем у него количество итераций для 2 алгоритмов и запускаем режим однократного запуска алгоритмов

**21-53:** если был выбран 2 режим, то берем у пользователя поток, куда записывать статистику, алгоритм, минимальное и максимальное количество итераций, количество шагов и запускаем режим накопления статистики

Все модули программы были описаны сверху, теперь перейдем к тому, как использовать данную программу.

**3.4 ПОРЯДОК ЭКСПЛУАТАЦИИ ПРОГРАММЫ**

Сначала нужно записать координаты всех точек, по которым нужно пройти в файл points, который находится в директории программы в таком виде: (x, y).

Затем запустить программу. Будет предоставлен выбор режима использования программы (рисунок 1).

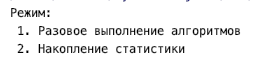
****

Рисунок 1 - Выбор режима

Если выбран первый режим, то нужно будет выбрать количество итераций для *случайного* и *генетического алгоритма* (рисунок 2).

****

Рисунок 2 - задание кол-ва итераций

Затем будет вывод процесса выполнения алгоритмов и вывод результаты выполнения алгоритмов (рисунок 3) и на этом программа завершится.

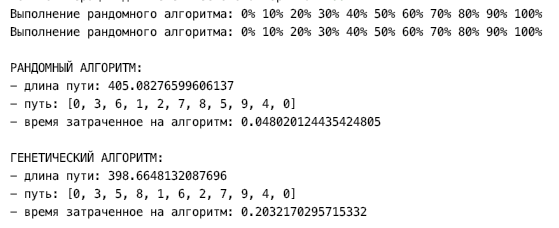


Рисунок 3- вывод результатов

Если же выбран второй режим, будет предложено выбрать куда выводить статистику (рисунок 4).

****

Рисунок 4 - выбор потока

Если был выбран второй пункт, то еще необходимо ввести имя файла (рисунок 5).

****

Рисунок 5 - указание файла

Затем будет предложено выбрать алгоритм, для накопления статистики (рисунок 6).

****

Рисунок 6 - выбор алгоритма

Далее выбор минимального и максимального количества итераций и количество шагов (рисунок 7).



Рисунок 7 - задание параметров

После чего проходит несколько выполнений алгоритма и запись статистики в файл или вывод на экран (рисунок 8) и на этом программ завершит свою работу.

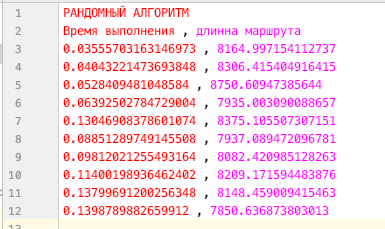


Рисунок 8 - вывод статистики

**4 СРАВНЕНИЕ ЭФЕКТИВНОСТИ ГЕНЕТИЧЕСКОГО АЛГОРИТМА**

Сейчас понадобится режим накопления статистики в реализованной программе: с помощью него можно очень просто получить большие объемы статистических данных в удобном виде для построения таблиц и проведения аналитики над ними.

Будет проведено 4 исследования: 1) сравнение результата, полученного с помощью *генетического алгоритма,* с эталонным и единственным правильным результатом, полученным с помощью полного перебора; сравнение эффективности *генетического алгоритма* со *случайным алгоритмом* на 2) малом количестве входных данных (20 точек), 3) среднем количестве входных данных (50 точек), 4) большом количестве входных данных (100 точек).

1. Сравнение результата, полученного с помощью *генетического алгоритма,* и значения, полученного полным перебором показало, что разработанный алгоритм работает корректно и дает правильный ответ при малом количестве городов. При большом количестве городов (более 9) не удается высчитать наилучший маршрут полным перебором, так как для этого потребуется большое количество вычислительных ресурсов. Вот пример: результат, полученный с помощью перебора (рисунок 9) и результат, полученный с помощью *генетического алгоритма* (рисунок 10) одинаковы.

****

Рисунок 9 - Результат перебора

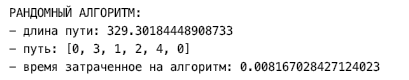
****

Рисунок 10 - Результат генетического алгоритма

2. Сравнение эффективности *генетического алгоритма* и *случайного алгоритма* при малом объеме входных значений.

Как видно из графика (рисунок 11), генетический алгоритм лидирует практически всегда, а когда не лидирует – идет вровень со *случайным алгоритмом*. То есть, на малых объемах входных данных *генетический алгоритм* эффективней, чем случайный, в среднем на 15 процентов.

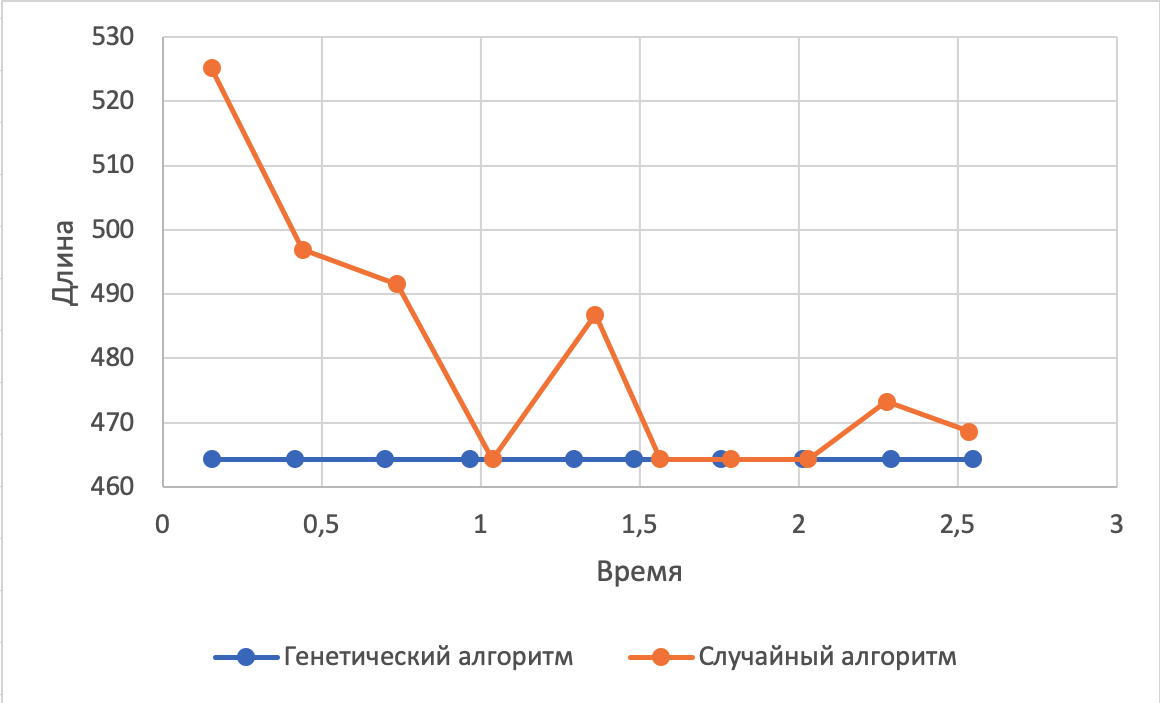


Рисунок 11 - Малый объем данных

3. Сравнение эффективности *генетического алгоритма* и *случайного алгоритма* при среднем объеме входных значений.

На предоставленном графике (рисунок 12), на среднем количестве входных данных *генетический алгоритм* ведет себя еще лучше, чем на малых, по сравнению со *случайным алгоритмом.*

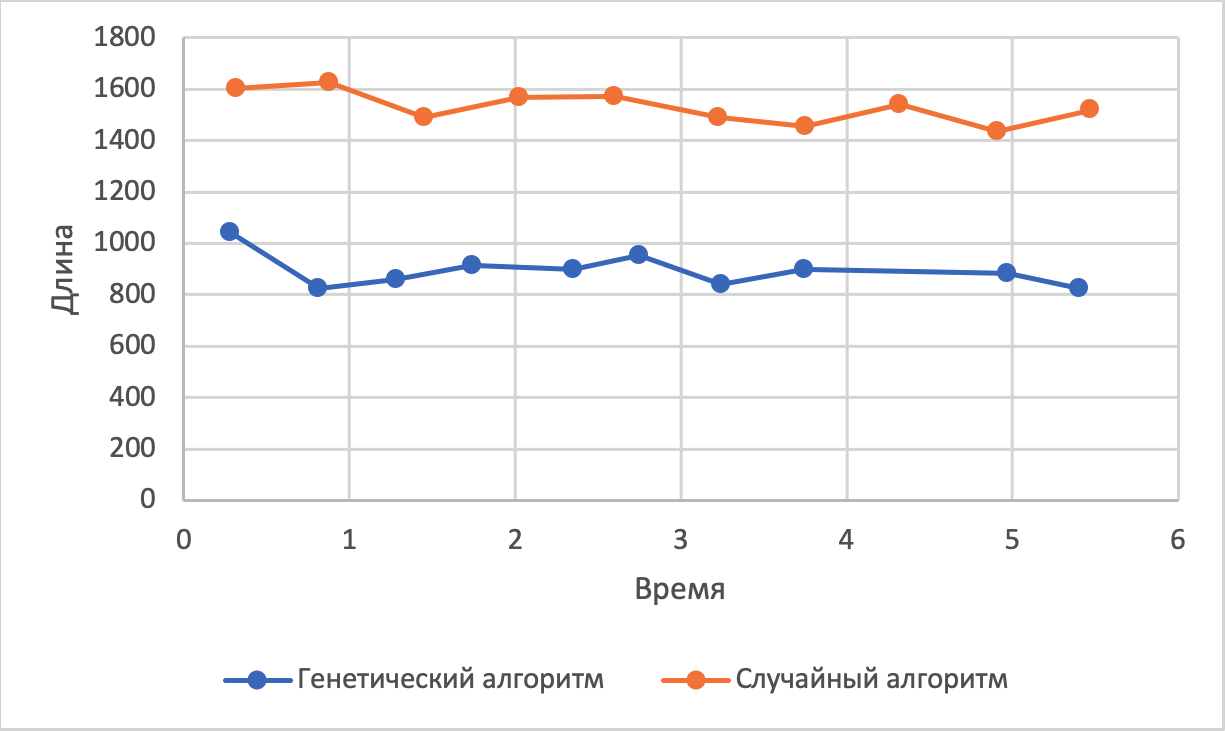


Рисунок 12 - Средний объем данных

4. Сравнение эффективности *генетического алгоритма* и *случайного алгоритма* при большом объеме входных значений.

Как и следовало ожидать, из графика (рисунок 13) отчетливо видно, что *генетический алгоритм,* при любом времени выполнения алгоритма, эффективней чем *случайный алгоритм*

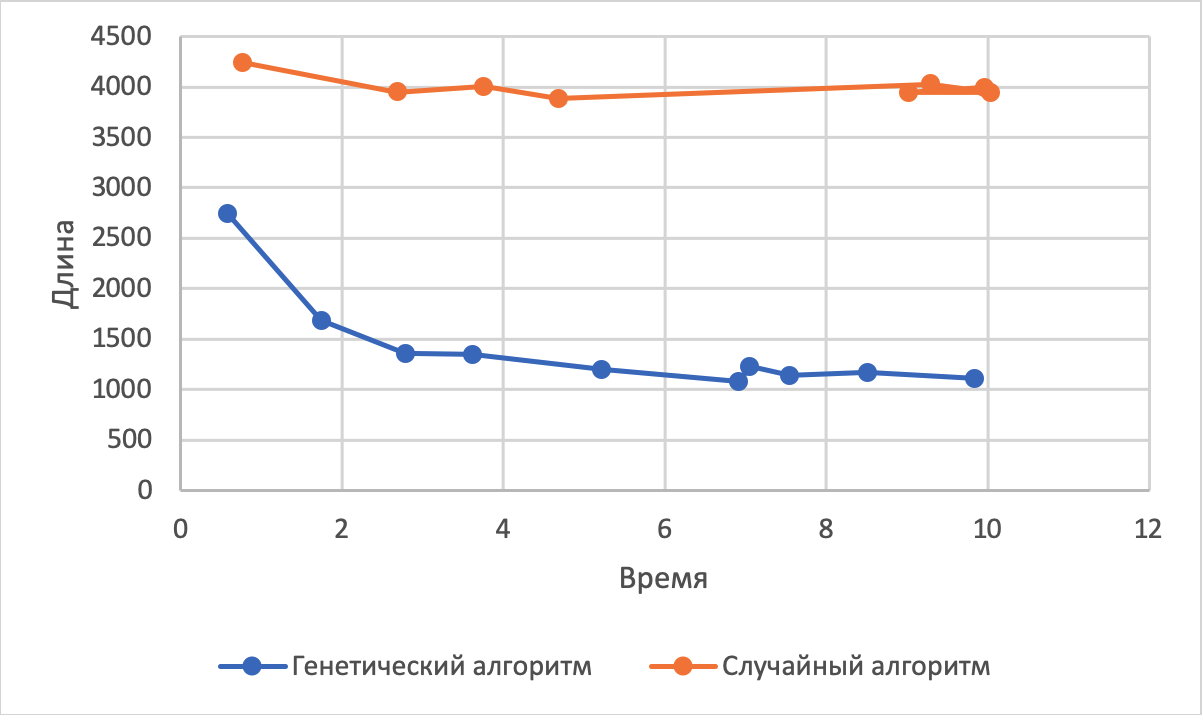
****

Рисунок 13 - Большой объем данных

Если смотреть дальше, на результаты работы этих алгоритмов при еще больших входных данных, то, закономерно, *генетический алгоритм* всегда будет эффективней *случайного алгоритма*. Остановимся на рассмотри объема входных данный в 100 точек, так как дальше все будет аналогично.

*Генетический алгоритм* очень хорошо показал себя в этих сравнениях и в сравнении с истинным значением, полученного с помощью перебора всех возможных маршрутов, с точки зрения эффективности по времени и точности подобранного маршрута.



**ЗАКЛЮЧЕНИЕ**

В данной курсовой работы были рассмотрены и описаны точные алгоритмы, такие как: *алгоритм полного перебора* и *метод ветвей и границ.* Так же были рассмотрены и описаны эвристические алгоритмы, такие как: *случайный метод,* *метод включения дальнего (МВД),* *BV-метод, муравьиный алгоритм* и *генетический алгоритм.*

В процессе написания работы приняты в рассмотрение достоинства и недостатки этих групп алгоритмов, а также достоинства и недостатки отдельно каждого алгоритма сортировки.

Более подробно, по сравнению с остальными, рассмотрен *генетический алгоритм* и то, как с помощью него можно решить поставленную *задачу коммивояжера.*

На втором этапе, то есть на программной реализации, была разработана программа, которая позволяет решать *задачу коммивояжера* с помощью *генетического алгоритма* и *случайного алгоритма*, выводит время для выполнения алгоритма, а также имеет режим накопления статистики, для анализа работы этих алгоритмов.

Написано руководство для пользователя о том, как пользоваться данной программой.

И самое главное, был проведен анализ работы *генетического алгоритма*. Произведен сравнительный анализ результата работы *генетического алгоритма* с результатом, полученным из полного перебора всех решений; произведен сравнительный анализ *генетического алгоритма* с *случайным алгоритмом* на малом объеме входных данных, на среднем объеме входных данных и большом объеме входных данных, во всех этих категориях *генетический алгоритм* показал себя лучше и только наращивал преимущество с увеличением объема входных данных.

*Генетический алгоритм* показал себя как состоятельный алгоритм, который может решать *задачу коммивояжера* с высокой точность, которая подходит для прикладных задач в реальной жизни, за небольшое время, что тоже очень важно. Его использовали, используют и будут использовать в решении *задачи коммивояжера*.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Образовательный портал habr : генетический алгоритм, просто о сложном : официальный сайт объединения «TM» - Москва, 2008-2019 – URL: [**https://habr.com/ru/post/128704/**](https://habr.com/ru/post/128704/) - Текст: электронный.

2. Образовательный портал py-algorithm : задачи оптимизации – URL: [**http://py-algorithm.blogspot.com/2014/10/blog-post\_21.html**](http://py-algorithm.blogspot.com/2014/10/blog-post_21.html) - Текст: электронный.

3. Интернет-энциклопедия : задача коммивояжера – URL: [**https://ru.wikipedia.org/wiki/Задача\_коммивояжёра**](https://ru.wikipedia.org/wiki/Задача_коммивояжёра) – Текст: электронный.

4. Вестник Астраханского государственного технического университета. Серия: Управление, вычислительная техника и информатика 2009, №2 : В. О. Боронзов. Исследование решения задачи коммивояжера – Текст: непосредственный.

5. Радиоэлектроника, информатика, управление 2003, : В. М. Гавриляко. задача коммивояжера и генетические алгоритмы – Текст: непосредственный.

**Приложение А**

**random\_optimization.py :**

**from** typing **import** Callable, List, Tuple  
**from** random **import** choice  
  
  
**def** random\_optimization(points: [(int, int)],  
 cost\_func: Callable[[List[Tuple[int, int]], List[int]], int],  
 max\_iter: int = 100000,  
 log: bool = **False**) -> (int, int):  
 *"""  
 points - список координат точек, через которые нужно пройти  
 cost\_func - функция, высчитывающая длину маршрута  
 max\_iter - максимальное число генерирования нового случаного решения  
 log - производить ли логирование (процесс выполнения)  
 """* **if** log:  
 print(**'Выполнение рандомного алгоритма:'**, end=**' '**)  
  
 loading\_line = max\_iter / 10  
 percent = 0  
  
 best\_length = 9999999999  
 best\_path = **None  
  
 for** i **in** range(0, max\_iter):  
 **if** i % loading\_line == 0 **and** log:  
 print(percent, **'%'**, sep=**''**, end=**' '**)  
 percent += 10  
  
 *# выбор отправной точки (всегда первая точка) и удаляем ее из оставшихся точек* remaining\_points = [i **for** i **in** range(0, len(points))]  
 begin\_point = remaining\_points[0]  
 remaining\_points.remove(begin\_point)  
  
 *# выбор случайного решения* path = [begin\_point]  
 **while** len(remaining\_points):  
 rand\_point = choice(remaining\_points)  
 path.append(rand\_point)  
 remaining\_points.remove(rand\_point)  
 path.append(begin\_point) *# маршрут должен завершаться в начальной точке* length = cost\_func(points, path)  
 **if** length < best\_length:  
 best\_length = length  
 best\_path = path  
  
 **if** log:  
 *# для перевода строки после логирования* print(percent, **'%'**, sep=**''**)  
  
 **return** best\_length, best\_path

**Приложение Б**

**genetic\_optimization.py :**

**from** typing **import** Callable, List, Tuple  
**from** random **import** randint, choice, random  
  
  
**def** genetic\_optimization(domain: [(int, int)],  
 cost\_func: Callable[[List[Tuple[int, int]], List[int]], int],  
 pop\_size: int = 100,  
 mut\_prob: int = 0.3,  
 elite: float = 0.2,  
 max\_iter: int = 1000,  
 log: bool = **False**) -> (int, int):  
 *"""  
 domain - список координат точек, через которые нужно пройти  
 cost\_func - функция, высчитывающая длину маршрута  
 pop\_size - размер популяции  
 mut\_prob - чем меньше, тем реже берем мутацию и чаще скрещивание  
 elite - доля особей в популяции, считающихся хорошими решениями и переходящих в следующее поколение  
 max\_iter - количество поколений  
 log - производить ли логирование (процесс выполнения)  
 """* **if** log:  
 print(**'Выполнение рандомного алгоритма:'**, end=**' '**)  
  
 *# мутация* **def** mutate(vec):  
 i = randint(1, len(domain)-2)  
 **return** vec[:i] + [vec[i+1]] + [vec[i]] + vec[i+2:]  
  
 *# скрещивание (из одной особи)* **def** crossover(vec):  
 begin = randint(1, len(domain)-4)  
 end = randint(begin, len(domain)-1)  
 **return** vec[:begin] + vec[begin:end][::-1] + vec[end:]  
  
 *# создаем первую популяция* pop = []  
 **for** \_ **in** range(pop\_size):  
 *# выбор отправной точки (всегда первая точка) и удаляем ее из оставшихся точек* remaining\_gens = [i **for** i **in** range(0, len(domain))]  
 begin\_gen = remaining\_gens[0]  
 remaining\_gens.remove(begin\_gen)  
  
 *# составление хромосомы из случайных генов* vec = [begin\_gen]  
 **while** len(remaining\_gens):  
 rand\_gen = choice(remaining\_gens)  
 vec.append(rand\_gen)  
 remaining\_gens.remove(rand\_gen)  
 vec.append(begin\_gen) *# маршрут должен завершаться в начальной точке* pop.append(vec)  
  
 *# Сколько лучших оставляем из каждого поколения* top\_elite = int(elite \* pop\_size)  
  
 loading\_line = max\_iter / 10  
 percent = 0  
  
 *# входим в основной цикл генетического алгоритма* **for** i **in** range(max\_iter):  
 **if** i % loading\_line == 0 **and** log:  
 print(percent, **'%'**, sep=**''**, end=**' '**)  
 percent += 10  
  
 scores = [(cost\_func(domain, v), v) **for** v **in** pop] *# (длина пути, путь)* scores.sort()  
 ranked = [v **for** (s, v) **in** scores] *# (путь)  
  
 # Сначала включаем только победителеи* pop = ranked[0:top\_elite]  
  
 *# Добавляем особей, полученных мутацией и скрещиванием родителей* **while** len(pop) < pop\_size:  
  
 i = randint(0, pop\_size - 1)  
 **if** random() < mut\_prob:  
 *# Мутация* pop.append(mutate(ranked[i]))  
 **else**:  
 *# Скрещивание* pop.append(crossover(ranked[i]))  
  
 scores = [(cost\_func(domain, v), v) **for** v **in** pop]  
 scores.sort()  
  
 **if** log:  
 *# для перевода строки после логирования* print(percent, **'%'**, sep=**''**)  
  
 *# возвращаем первое место* **return** scores[0][0], scores[0][1]

**Приложение В**

**functions.py :**

**from** math **import** sqrt  
**from** typing **import** Callable, List, Tuple  
**from** re **import** findall  
**from** time **import** time  
  
  
**def** distance\_between(point1: (int, int),  
 point2: (int, int)) -> float:  
 *"""расстояние между 2 точками"""* distance = sqrt(abs(point1[0] - point2[0])\*\*2 + abs(point1[1] - point2[1])\*\*2)  
 **return** distance  
  
  
**def** calculate\_path\_length(points: List[Tuple[int, int]],  
 path: [int]) -> float:  
 *"""высчитывание общей длины маршрута"""* length = 0  
 **for** i **in** range(0, len(path)-1):  
 length += distance\_between(points[path[i]], points[path[i+1]])  
 **return** length  
  
  
**def** load\_from\_file(filename: str) -> [(int, int)]:  
 *"""загрузка точек из файла"""* file = open(filename)  
 *# строки такого вида '12, 32', '23,44'* strings = findall(**r'(\s\*\d+\s\*,\s\*\d+\s\*)'**, file.read())  
  
 result = []  
 **for** point **in** strings:  
 *# теперь разбиваем строки на 2 числа* x\_and\_y = findall(**r'\d+'**, point)  
 *# и добавляем кортежом в массив* result.append((int(x\_and\_y[0]), int(x\_and\_y[1])))  
  
 **return** result  
  
  
**def** get\_length\_path\_duration(points: [(int, int)],  
 cost\_func: Callable,  
 opt\_func: Callable,  
 max\_iter\_: int,  
 log\_: bool) -> (float, [int], float):  
 *"""возвращает длину, путь и время затраченное на алгоритм"""* start\_ = time()  
 result\_length, result\_path = opt\_func(points, cost\_func, max\_iter=max\_iter\_, log=log\_)  
 end\_ = time()  
 duration\_ = end\_ - start\_  
 **return** result\_length, result\_path, duration\_  
  
  
**def** show\_result(name: str,  
 length: float,  
 path: float,  
 duration\_opt: float) -> **None**:  
 *"""вывести результат работы алгоритма"""* print()  
 print(name, **':'**, sep=**''**)  
 print(**'- длина пути:'**, length)  
 print(**'- путь:'**, path)  
 print(**'- время затраченное на алгоритм:'**, duration\_opt)

**Приложение Г**

**modes.py :**

**from** optimization.genetic\_optimization **import** genetic\_optimization  
**from** optimization.random\_optimization **import** random\_optimization  
**from** functions **import** (get\_length\_path\_duration, calculate\_path\_length, show\_result)  
**from** typing **import** Callable  
**from** sys **import** stdout  
  
  
**def** run\_once(points: list, iter\_rand\_: int, iter\_gen\_: int) -> **None**:  
 *"""выполнить оба алгоритма один раз  
 points - список точек  
 iter\_rand\_ - кол-во итераций рандомного алгоритма  
 iter\_gen\_ - кол-во итераций генетического алгоритма  
 """* rand\_length, rand\_path, duration\_rand\_opt = get\_length\_path\_duration(points, calculate\_path\_length,random\_optimization, max\_iter\_=iter\_rand\_, log\_=**True**)

gen\_length, gen\_path, duration\_gen\_opt = get\_length\_path\_duration(points, calculate\_path\_length, genetic\_optimization, max\_iter\_=iter\_gen\_, log\_=**True**)

show\_result(**'РАНДОМНЫЙ АЛГОРИТМ'**, rand\_length, rand\_path, duration\_rand\_opt)  
 show\_result(**'ГЕНЕТИЧЕСКИЙ АЛГОРИТМ'**, gen\_length, gen\_path, duration\_gen\_opt)  
  
  
**def** statistics\_accumulation(points: list, opt\_: Callable, name\_: str,  
 min\_iter\_: int, max\_iter\_: int, count\_gaps\_: int, file\_=stdout) -> **None**:  
 *"""Запись статистики (время / найденный путь)  
 points - список точек  
 opt\_ - функция оптимизации  
 name\_ - имя функции оптимизации  
 min\_iter - минимальное кол-во итераций алгоритма  
 max\_iter - минимальное кол-во итераций алгоритма  
 count\_gaps\_ - кол-во интервалов  
 file\_ - файл для записи статистики  
 """* print(**'выполнение...'**)  
 print(name\_, file=file\_)  
 step\_ = (max\_iter\_ - min\_iter\_) // count\_gaps\_  
  
 **for** iter\_ **in** range(min\_iter\_, max\_iter\_, step\_):  
 length, path, duration = get\_length\_path\_duration(points, calculate\_path\_length, opt\_, max\_iter\_=iter\_, log\_=**False**)  
 print(duration, **','**, length, file=file\_)

**Приложение Д**

**main.py :**

**from** optimization.random\_optimization **import** random\_optimization  
**from** optimization.genetic\_optimization **import** genetic\_optimization  
**from** functions **import** load\_from\_file  
**from** modes **import** run\_once, statistics\_accumulation  
**from** sys **import** stdout  
**from** re **import** findall, search  
  
  
**def** main():  
 all\_points = load\_from\_file(**'points'**)  
  
 answer = input(**'Режим:\n 1. Разовое выполнение алгоритмов\n 2. Накопление статистики\n'**)  
  
 **if** answer == **'1'**:  
 iter\_rand = int(input(**'кол-во итераций для рандомного алгоритма: '**))  
 iter\_gen = int(input(**'кол-во итераций для генетического алгоритма: '**))  
  
 run\_once(all\_points, iter\_rand, iter\_gen)  
 **elif** answer == **'2'**:  
 answer = input(**'Запись статистики:\n 1. Консоль\n 2. Файл\n'**)  
 **if** answer == **'1'**:  
 stream = stdout  
 **elif** answer == **'2'**:  
 stream = open(input(**'Файл: '**), **'w'**)  
  
 answer = input(**' 1. Рандомный алгоритм\n 2. Генетический алгоритм\n'**)  
 **if** answer == **'1'**:  
 name = **'РАНДОМНЫЙ АЛГОРИТМ'** opt = random\_optimization  
 **elif** answer == **'2'**:  
 name = **'ГЕНЕТИЧЕСКИЙ АЛГОРИТМ'** opt = genetic\_optimization  
  
 answer = input(**'минимальное - максимальное кол-во итераций: '**)  
 answer = search(**r'\s\*\d+\s\*-\*\s\*\d+\s\*'**, answer).group(0)  
 delta = findall(**r'\d+'**, answer)  
  
 **if** len(delta) != 2:  
 exit(**'должно быть 2 значения'**)  
  
 min\_iter, max\_iter = int(delta[0]), int(delta[1])  
 **if** min\_iter > max\_iter:  
 min\_iter, max\_iter = max\_iter, min\_iter  
  
 answer = input(**'кол-во промежутков: '**)  
 *# на случай, если пользователь введет не одно число, тогда возьмем только первое* count\_gaps = int(search(**r'\d+'**, answer).group(0))  
  
 statistics\_accumulation(all\_points, opt, name, min\_iter, max\_iter, count\_gaps, stream)  
  
  
**if** \_\_name\_\_ == **'\_\_main\_\_'**:  
 main()