Федеральное государственное бюджетное образовательное учреждение

высшего образования

«Сибирский государственный университет телекоммуникаций и

информатики»

(СибГУТИ)

02.03.02 Фундаментальная информатика и информационные технологии

Профиль: Системное программное обеспечение

(очная форма обучения)

ОТЧЕТ ПО учебной ПРАКТИКЕ

на кафедре вычислительных систем

(наименование структурного подразделения СибГУТИ)

ГЕНЕТИЧЕСКИЙ АЛГОРИТМ. РЕАЛИЗАЦИЯ И ОПИСАНИЕ

Выполнил:

студент института ИВТ

гр. ИС-241 / Кулик П.Е./

«25» мая 2024г. (подпись)

Проверил:

Руководитель от СибГУТИ / Перышкова Е.Н./

«25» мая 2024г. (подпись)

Новосибирск 2024

**План-график проведения**  учебной **практики**

Вид практики

Кулика Павла Евгеньевича

Фамилия Имя Отчество студента

института Информатика и вычислительная техника , 2 курса,

гр. ИС-241

Направление: 02.03.02 Фундаментальная информатика и информационные технологии

Код – Наименование направления (специальности)

Профиль: Системное программное обеспечение

Место прохождения практики кафедра вычислительных систем

Объем практики: **108/3** часов/ЗЕ

Вид практики  ***учебная***

Тип практики ***научно-исследовательская работа (получение первичных навыков научно-исследовательской работы)***

Срок практики с "29" января 2024 г.

по "25" мая 2024 г.

Содержание практики\*:

|  |  |
| --- | --- |
| Наименование видов деятельности | Дата (начало – окончание) |
| 1. Общее ознакомление со структурным подразделением предприятия, вводный инструктаж по технике безопасности | 29.01.2024–01.02.2024 |
| 2. Выдача задания на практику, деление студентов на группы (если необходимо), определение конкретной индивидуальной темы, формирование плана работ | 02.02.2024–04.02.2024 |
| 3. Работа с библиотечными фондами структурного подразделения или предприятия, сбор и анализ материалов по теме практики | 06.02.2024–11.02.2024 |
| 4. Выполнение работ в соответствии с составленным планом:  – Реализация алгоритма  – Подготовка тестовой программы  – Сбор экспериментальных данных | 13.02.2024 – 20.05.2024 |
| 5. Анализ полученных результатов и произведенной работы  Составление отчета по практике, защита отчета | 22.05.2024–25.05.2024 |

\*В соответствии с программой практики

Руководитель от СибГУТИ / Перышкова Е.Н./

«29» \_\_\_\_01\_\_\_\_ 2024г. (подпись)

**ОГЛАВЛЕНИЕ**

[ЗАДАНИЕ НА ПРАКТИКУ 4](#_Toc167198494)

[ВВЕДЕНИЕ 5](#_Toc167198495)

[ОСНОВНАЯ ЧАСТЬ 6](#_Toc167198496)

[Постановка задачи 6](#_Toc167198497)

[Использованная терминология 6](#_Toc167198498)

[Алгоритм решения задачи 8](#_Toc167198499)

[Практическое исследование алгоритма 9](#_Toc167198500)

[Сходимость алгоритма 10](#_Toc167198501)

[Эффективность алгоритма 12](#_Toc167198502)

[Общая оценка алгоритма 15](#_Toc167198503)

[ЗАКЛЮЧЕНИЕ 15](#_Toc167198504)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 16](#_Toc167198505)

[ПРИЛОЖЕНИЕ 17](#_Toc167198506)

[Отзыв о работе студента 26](#_Toc167198507)

[Уровень освоения компетенций 27](#_Toc167198508)

# ЗАДАНИЕ НА ПРАКТИКУ

В соответствии с темой практики задание на практику заключается в том, чтобы реализовать и описать генетический алгоритм.

Сама реализация происходит в несколько этапов:

1. Поиск и изучение теоретической информации (06.02.2024–11.02.2024);
2. Реализация алгоритма (13.02.2024 – 13.04.2024);
3. Подготовка тестовой программы для сбора экспериментальных данных и сбор данных для анализа и построения графиков (14.04.2024 – 20.05.2024);
4. Составление отчёта о проделанной работе (22.05.2024–25.05.2024).

# ВВЕДЕНИЕ

Первые попытки создания алгоритма, имитирующего эволюцию живых организмов, принадлежат Нильсу Баричелли, но наиболее широкое распространение этот алгоритм получил после публикации книги Джона Холланда «Адаптация в естественных и искусственных системах» в 1975 году. Но только спустя 10 лет генетический алгоритм перешёл из стадии теоретического исследования к развитию его практических применений.

Основная идея генетического алгоритма (ГА) заключается в имитации некоторых принципов живой природы, изложенных в эволюционной теории Чарльза Дарвина. Для описания ГА были заимствованы некоторые термины, ранее применявшиеся для описания живой природы, такие как особь, популяция, скрещивание и так далее. С точки зрения описания ГА все эти термины используются для названия операндов и операций, используемых для решения задач поиска и оптимизации.

# ОСНОВНАЯ ЧАСТЬ

Было принято решение реализовать ГА, решающий какую-либо практическую задачу, а потом исследовать его эффективность. В качестве задачи для решения была выбрана задача коммивояжёра, которая заключается в поиске самого выгодного маршрута, проходящего через указанные города хотя бы один раз с последующим возвращением в исходный город.

## Постановка задачи

Имеется область, на которой располагается n городов, являющихся вершинами vi, взвешенного графа A, у которого каждая вершина соединена со всеми остальными при помощи рёбер ek, с весом wk ≥ 1. При этом у каждой вершины есть как минимум 2 ребра с весом равным единице, соединяющих её с какими-либо двумя другими вершинами, благодаря чему кратчайший путь между всеми вершинами с последующим возвращением в изначальную равен по длине n. Требуется найти последовательность вершин такую, которая начинается и заканчивается в первой вершине и содержит в себе все остальные вершины графа хотя бы по одному разу, при этом между каждой парой соседних вершин вес ребра равен единице, либо же хотя бы приблизиться к ней.

## Использованная терминология

Для решения поставленной задачи был использован генетический алгоритм. Терминология в рамках поставленной задачи следующая:

- Ген – число gj, соответствующее номеру i вершины vi графа A;

- Геном – вектор G={gj} - упорядоченная последовательность уникальных генов, которая представляет из себя путь из первой вершины через все остальные с последующим возвращением в первую, при этом содержащий в себе только одну первую вершину;

- Начальный геном – геном, в котором все гены расположены по возрастанию;

- Особь – это объект d, содержащий в себе геном и набор методов для работы с ним;

- Приспособленность особи – это длина пути, который составляет геном или же сумма длин рёбер графа, располагающихся между вершинами, на которые указывают соседние гены в геноме с добавлением длины ребра между вершинами, на которые указывают первый и последний ген;

- Популяция – вектор I={di} из нескольких особей, упорядоченных по неубыванию приспособленности в количестве, заданном пользователем;

- Мутация – функция, которая меняет местами два случайно выбранных гена в геноме особи;

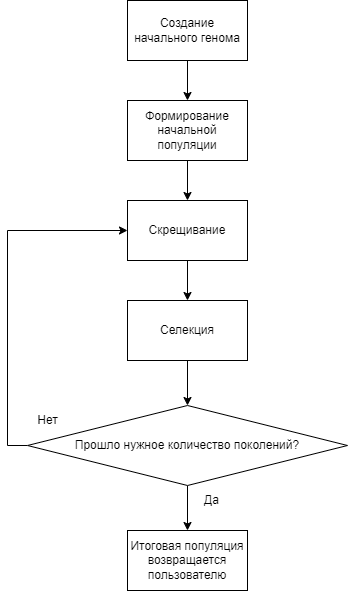
- Начальная популяция – популяция, сгенерированная на основе заданного начального генома;

- Скрещивание – функция, которая принимает в качестве аргументов вектор I и число 0 ≤ m ≤ 100, представляющее из себя шанс возникновения мутации у новых особей в процентах. Функция выбирает две особи из популяции случайным образом, после чего случайным образом выбирает опорную точку (номер гена), вокруг которой будет происходить обмен генами для генерации новых особей. За этим следует генерация двух новых особей по следующей схеме: первая из двух новых особей получает все гены первого родителя до опорной точки, после чего получает те гены второго родителя, которые начинаются с опорной точки и не совпадают с теми генами, которые уже попали в новую особь. Далее, если генов не хватило для полного генома, то добавляются гены из первой родительской особи такие, которые ещё не были добавлены до этого. Вторая особь формируется аналогично зеркальным образом. После того, как две новые особи были сформированы происходит мутация с указанным шансом и новые особи добавляются в конец популяции;

- Селекция – функция, которая принимает в качестве аргументов вектор особей и размер итоговой популяции, сортирует особей по их приспособленности и отбрасывает тех особей, которые имеют наибольшие значения приспособленности и выходят за рамки заданного количества особей, тем самым формируя популяцию.

## Алгоритм решения задачи

Сам алгоритм решения задачи состоит из небольшого количества шагов (рис. 1). Первым делом формируется начальный геном, на основе которого создаётся начальная популяция. Далее на протяжении заданного количества поколений происходят скрещивание и селекция, после чего получившаяся популяция становится доступна пользователю.



*Рисунок 1*. Блок-схема ГА.

## Практическое исследование алгоритма

Тестирование работы алгоритма происходило на вычислительном кластере F (Oak) СибГУТИ.

Конфигурация кластерной ВС Oak:

Кластер Oak укомплектован 6 вычислительными узлами, управляющим узлом, вычислительной и сервисной сетями связи, а также системой бесперебойного электропитания.

Узлы построены на базе серверной платформы Intel S5520UR. На каждом узле размещено два четырёхядерных процессора Intel Xeon E5620 с тактовой частотой 2,4 GHz. Пиковая производительность кластера – 460 GFLOPS.

Конфигурация вычислительного узла:

|  |  |
| --- | --- |
| Системная плата | Intel S5520UR |
| Процессор | 2 x Intel Xeon E5620 (2,4 GHz; Intel-64) |
| Оперативная память | 24 GB (6 x 4GB DDR3 1067 MHz) |
| Жесткий диск | SATAII 250GB (Seagate Barracuda ST3250318AS) |
| Сетевая карта | 1 x Mellanox MT26428 InfiniBand QDR 2 x Intel Gigabit Ethernet (Intel 82575EB Gigabit Ethernet Controller) |
| Корпус | Rack mount 2U |

Конфигурация управляющего узла:

|  |  |
| --- | --- |
| Системная плата | Intel S5520UR |
| Процессор | 2 x Intel Xeon E5620 (2,4 GHz; Intel-64) |
| Оперативная память | 24 GB (6 x 4GB DDR3 1067 MHz) |
| Жесткий диск | 4 x SATAII 500 GB (Seagate Barracuda ST3500514NS) |
| Сетевая карта | 1 x Mellanox MT26428 InfiniBand QDR 2 x Intel Gigabit Ethernet (Intel 82575EB Gigabit Ethernet Controller) |
| Корпус | Rack mount 2U |

Конфигурация коммуникационной среды:

|  |  |
| --- | --- |
| Сервисная сеть | Коммутатор Gigabit Ethernet (D-Link DGS-1224T) |
| Вычислительная сеть | Коммутатор Infiniband QDR (Mellanox InfiniScale IV IS5030 QDR 36-Port InfiniBand Switch) |

Все замеры производились с использованием системы пакетной обработки заданий. Сам алгоритм был реализован с использованием языка C++ и компилятора GNU GCC.

### Сходимость алгоритма

Сходимость алгоритма зависит от процента мутаций, количества поколений и размера популяции. Для исследования алгоритма вводится такой параметр как точность, который вычисляется следующим образом: . Значение 100% означает, что был найден кратчайший маршрут. Любое значение, отличное от 100% означает, что кратчайший маршрут не был найден. Чем значение меньше, тем найденный маршрут длиннее кратчайшего.

Мутации в ходе скрещивания имеют большое значение с точки зрения сходимости алгоритма, что можно наглядно увидеть на графике (рис. 2). При одном и том же количестве поколений и городов точность выполнения алгоритма может отличаться на 50% в зависимости от выбранного процента мутаций.



*Рисунок 2*. Сходимость алгоритма при разном проценте мутаций.

Количество поколений очевидным образом влияет на сходимость алгоритма, так как чем больше скрещиваний происходит, тем больше различных вариантов маршрутов возникает. Но чем короче становится лучший найденный маршрут, тем сложнее становится искать более короткий, что и видно на графике (рис. 3).



*Рисунок 3*. Сходимость алгоритма при разном количестве поколений.

Чрезмерное увеличение размера популяции исключительно негативно сказывается на сходимости алгоритма, что наглядно видно на графике (рис. 4). Не трудно сделать вывод, что нет никакого смысла устанавливать размер популяции более 5 особей. Это станет ещё очевиднее после оценки эффективности алгоритма.



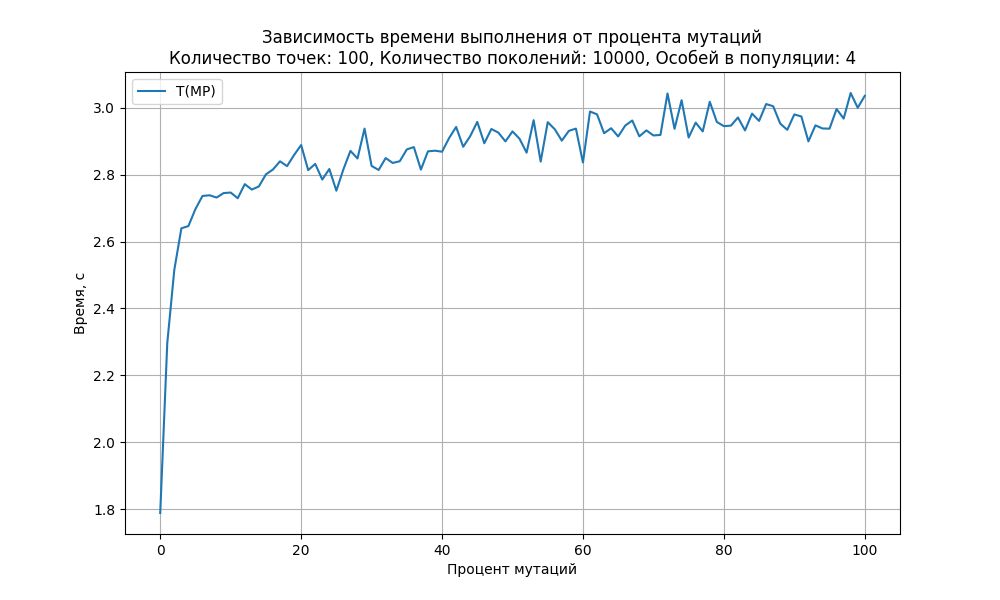
*Рисунок 5*. Сходимость алгоритма при разных размерах популяции.

### Эффективность алгоритма

Исследовалась зависимость эффективности алгоритма от следующих параметров:

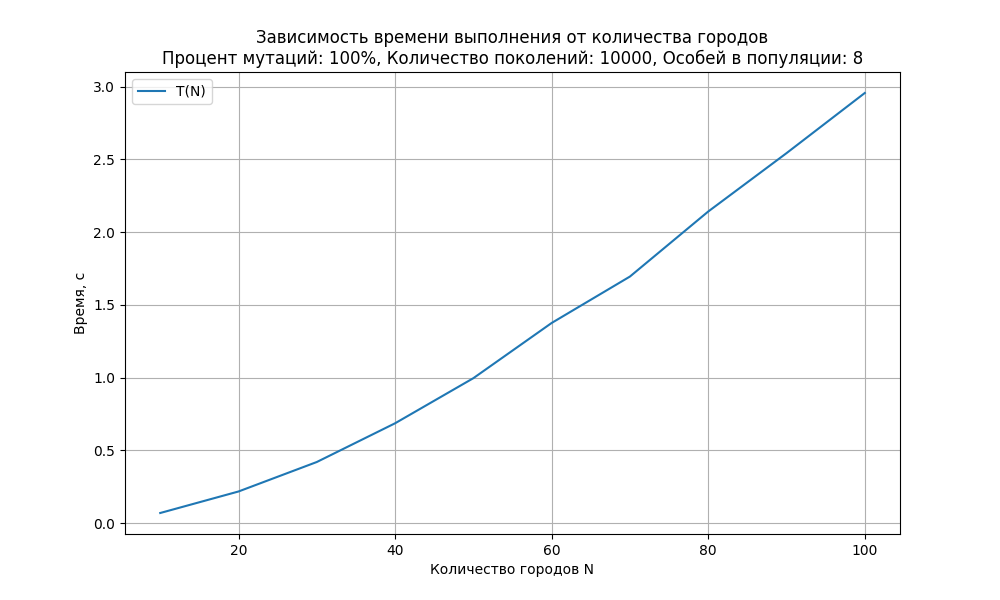
1. Процент мутаций;
2. Количество городов или же количество вершин графа;
3. Размер популяции.

Изменение количества мутаций не сильно сказывается на времени выполнения алгоритма (рис. 6), за исключением значений, близких к нулю, но тест сходимости показал, что такие значения лучше не использовать так как они негативно сказываются на скорости поиска оптимального решения. Сложность алгоритма с точки зрения процента мутаций O(log P), где P – процент мутаций.



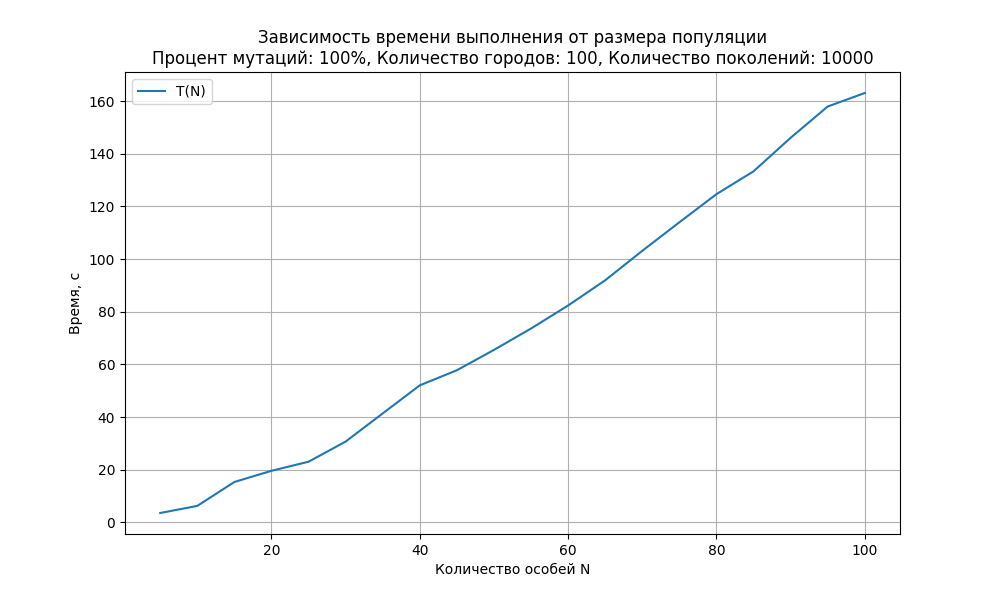
*Рисунок 6.* Зависимость времени выполнения от процента мутаций.

Увеличение количества городов линейно влияет на сложность алгоритма (рис. 7), так как в функции скрещивания есть несколько циклов, которые просто осуществляют обмен генами, которых становится больше с увеличением количества городов, а в функции селекции происходит сортировка, для которой для каждой особи линейно вычисляется её приспособленность. Так как количество особей небольшое, то сам процесс сортировки, имеющий сложность O(log N), не вносит большого вклада. Таким образом, сложность алгоритма O(N), где N – количество городов.



*Рисунок 7.* Зависимость времени выполнения от количества городов.

Изменение размера популяции также линейно влияет на время выполнения алгоритма, как и изменение количества городов, но разница по времени выполнения между 5 особями и 100 слишком велика, а если учесть и то, что большое количество особей в популяции негативно влияет на точность алгоритма, то не остаётся никаких сомнений в том, что для корректной работы алгоритма не требуется более 5 особей в популяции (рис. 8).



*Рисунок 8.* Зависимость времени выполнения от размера популяции.

### Общая оценка алгоритма

ГА показывает неплохую эффективность в задаче поиска набора оптимальных решений, но поиск идеального решения задачи может занять значительное время при использовании ГА. Так как ГА опирается на псевдослучайные числа, то скорость поиска решения зависит не только от указанных выше параметров, но и от простого везения. Был проведён эксперимент, в котором был граф из 20 вершин и поиск решения за 1000 итераций или же 1000 поколений с 100% шансом мутаций. При таких условиях удалось найти путь длины 20 сначала за 2571 перезапуск алгоритма, потом за 20509 перезапусков, а в третий раз за 11465 перезапусков.

# ЗАКЛЮЧЕНИЕ

Таким образом, ГА выполняет поставленную задачу и предоставляет возможность получить множество различных оптимальных решений для одного и того же набора данных, но не гарантирует быстрое получение лучшего решения задачи. Имеет смысл попробовать модифицировать алгоритм с помощью, например, увеличения количества мутаций у каждой особи или другой схемы скрещивания особей. Возможно, в таком виде алгоритм будет работать эффективнее.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Ефимов А.В., Мамойленко С.Н., Перышкова Е.Н. Организация функционирования распределённых вычислительных систем при обработке наборов масштабируемых задач // Вестник томского государственного университета. 2011. № 2. С. 51-60.
2. Кудинов Ю. И. Интеллектуальные системы: учебное пособие. Липецк: ЛГТУ, 2014. 63 с.

# ПРИЛОЖЕНИЕ

Код программы, файл “main.cpp”

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195  196  197  198  199  200  201  202  203  204  205  206  207  208  209  210  211  212  213  214  215  216  217  218  219  220  221  222  223  224  225  226  227  228  229  230  231  232  233  234  235  236  237  238  239  240  241  242  243  244  245  246  247  248  249  250  251  252  253  254  255  256  257  258  259  260  261  262  263  264  265  266  267  268  269  270  271  272  273  274  275  276  277  278  279  280  281  282  283  284  285  286  287  288  289  290  291  292  293  294  295  296  297  298  299  300  301  302  303  304  305  306  307  308  309  310  311  312  313  314  315  316  317  318  319  320  321  322  323  324  325  326  327  328  329  330  331  332  333  334  335  336  337  338  339  340  341  342  343  344  345  346  347  348  349  350  351  352  353  354  355  356  357  358  359  360  361  362  363  364  365  366  367  368  369  370  371  372  373  374  375  376  377  378  379  380  381  382  383  384  385  386  387  388  389  390  391  392  393  394  395  396  397  398  399  400  401  402  403  404  405  406  407  408  409  410  411  412  413  414  415  416  417  418  419  420  421  422  423  424  425  426  427  428  429  430  431  432  433  434  435  436  437  438  439  440  441  442  443  444  445  446  447  448  449  450  451  452  453  454  455  456  457  458  459  460  461  462  463  464  465  466  467  468  469  470  471  472  473  474  475  476  477  478  479  480  481  482  483  484  485  486  487  488  489  490  491  492  493  494 | #include <algorithm>  #include <cstdlib>  #include <cstring>  #include <fstream>  #include <iomanip>  #include <iostream>  #include <map>  #include <string>  #include <sys/time.h>  #include <utility>  #include <vector>  **using** **namespace** std;  **enum** options {  n\_cities\_opt,  population\_size\_opt,  mutation\_percent\_opt,  n\_iterations\_opt  };  **double** **wtime**()  {  **struct** timeval t;  gettimeofday(&t, NULL);  **return** (**double**)t.tv\_sec + (**double**)t.tv\_usec \* **1E-6**;  }  **class** **city** {  **int** name;  map<**int**, **int**> roads;  **public:**  city(**int** Name = **0**) : name(Name)  {  }  **int** add\_road(**int** to, **int** length)  {  **if** (length < **1**)  **return** -**1**;  **if** (roads.count(to) == **1**)  **return** -**1**;  roads[to] = length;  **return** **0**;  }  **int** get\_road(**int** to)  {  **if** (roads.count(to) == **0**)  **return** -**1**;  **return** roads[to];  }  **void** set\_name(**int** new\_name)  {  name = new\_name;  }  **int** get\_name()  {  **return** name;  }  city& **operator**=(**const** city& other)  {  **this**->name = other.name;  **this**->roads = other.roads;  **return** \***this**;  }  };  **class** **area** {  vector<city> cities;  **public:**  area()  {  }  **int** add\_city(**int** new\_city\_name)  {  city new\_city(new\_city\_name);  **for** (**auto**& c : cities) {  **if** (c.get\_name() == new\_city\_name)  **return** -**1**;  }  cities.push\_back(new\_city);  **return** **0**;  }  **int** add\_city\_road(**int** a, **int** b, **int** length)  {  **if** (a == b || length < **1**)  **return** -**1**;  **int** a\_exist = **0**, b\_exist = **0**;  **int** a\_city\_name = a < b ? a : b;  **int** b\_city\_name = b > a ? b : a;  city a\_city;  **for** (**auto**& c : cities) {  **if** (c.get\_name() == a\_city\_name) {  a\_exist = **1**;  a\_city = c;  }  **if** (c.get\_name() == b\_city\_name) {  b\_exist = **1**;  }  }  **if** (!a\_exist || !b\_exist)  **return** -**1**;  **if** (a\_city.get\_road(b\_city\_name) != -**1**)  **return** -**1**;  a\_city.add\_road(b\_city\_name, length);  **for** (**auto**& c : cities)  **if** (c.get\_name() == a\_city\_name)  c = a\_city;  **return** **0**;  }  **int** get\_city\_road(**int** a, **int** b)  {  **if** (a == b)  **return** -**1**;  **int** a\_city\_name = a < b ? a : b;  **int** b\_city\_name = b > a ? b : a;  **for** (**auto**& c : cities) {  **if** (c.get\_name() == a\_city\_name)  **return** c.get\_road(b\_city\_name);  }  **return** -**1**;  }  **void** print()  {  **for** (**auto**& city\_a : cities)  **for** (**auto**& city\_b : cities) {  **if** (city\_a.get\_name() == city\_b.get\_name())  **continue**;  cout << city\_a.get\_name() << " --> " << city\_b.get\_name()  << " : "  << get\_city\_road(city\_a.get\_name(), city\_b.get\_name())  << endl;  }  }  **void** clear()  {  cities.clear();  }  **int** get\_size()  {  **return** cities.size();  }  };  area local\_area;  vector<**int**> path;  vector<**int**> cities\_list;  **class** **individual** {  vector<**int**> genome;  **int** genome\_size;  **public:**  individual() : genome\_size(**0**)  {  }  individual(vector<**int**> new\_genome, **int** size)  : genome(new\_genome), genome\_size(size)  {  }  **int** add\_chromosome(**int** chromosome)  {  **for** (**auto**& g : genome)  **if** (g == chromosome)  **return** -**1**;  genome.push\_back(chromosome);  genome\_size++;  **return** **0**;  }  vector<**int**> get\_genome()  {  **return** genome;  }  **void** make\_mutation()  {  **int** rand\_limit = genome\_size - **1**;  **int** a = rand() % rand\_limit + **1**;  **int** b = rand() % rand\_limit + **1**;  **while** (a == b)  b = rand() % rand\_limit + **1**;  swap(genome[a], genome[b]);  }  **int** size()  {  **return** genome\_size;  }  **size\_t** fitness() **const**  {  **size\_t** length = **0**;  **int** prev = -**1**;  **int** count = **0**;  **for** (**auto**& chromosome : genome) {  count++;  **if** (prev == -**1**) {  prev = chromosome;  **continue**;  }  length += local\_area.get\_city\_road(prev, chromosome);  prev = chromosome;  **if** (count == genome\_size)  length += local\_area.get\_city\_road(**0**, chromosome);  }  **return** length;  }  };  **bool** **operator**<(**const** individual& a, **const** individual& b)  {  **return** a.fitness() < b.fitness();  }  **void** crossing(vector<individual>& population, **int** mutation\_chance)  {  **int** population\_size = population.size();  **int** a\_number = rand() % population\_size;  **int** b\_number = rand() % population\_size;  **while** (a\_number == b\_number)  b\_number = rand() % population\_size;  vector<**int**> genome\_a = population[a\_number].get\_genome();  vector<**int**> genome\_b = population[b\_number].get\_genome();  **int** genome\_size = genome\_a.size();  **int** cross\_point = rand() % (genome\_size - **1**) + **1**;  individual child\_a;  individual child\_b;  **for** (**int** i = **0**; i < cross\_point; i++) {  child\_a.add\_chromosome(genome\_a[i]);  child\_b.add\_chromosome(genome\_b[i]);  }  **for** (**int** i = cross\_point; i < genome\_size; i++) {  child\_a.add\_chromosome(genome\_b[i]);  child\_b.add\_chromosome(genome\_a[i]);  }  **if** (child\_a.size() < genome\_size) {  **for** (**int** i = cross\_point; i < genome\_size; i++) {  child\_a.add\_chromosome(genome\_a[i]);  child\_b.add\_chromosome(genome\_b[i]);  }  }  **if** (rand() % **100** < mutation\_chance) {  child\_a.make\_mutation();  child\_b.make\_mutation();  }  population.push\_back(child\_a);  population.push\_back(child\_b);  }  **void** selection(vector<individual>& population, **size\_t** population\_size)  {  sort(population.begin(), population.end());  population.erase(population.begin() + population\_size, population.end());  }  vector<individual>  make\_initial\_population(**size\_t** population\_size, vector<**int**> genome)  {  individual adam(genome, genome.size());  individual **eva**(genome, genome.size());  eva.make\_mutation();  vector<individual> population = {adam, eva};  **while** (population.size() < population\_size)  crossing(population, **100**);  selection(population, population\_size);  **return** population;  }  vector<individual> find\_shortest\_path(  **int** population\_size,  vector<**int**> cities\_list,  **int** n\_iterations,  **int** mutation\_percent)  {  vector<individual> population  = make\_initial\_population(population\_size, cities\_list);  **for** (**int** i = **0**; i < n\_iterations; i++) {  crossing(population, mutation\_percent);  selection(population, population\_size);  }  **return** population;  }  **int** get\_right\_path(**size\_t** path\_len)  {  **if** (path\_len == path.size())  **return** -**1**;  path.clear();  cities\_list.clear();  **for** (**size\_t** i = **0**; i < path\_len; i++) {  path.push\_back(i);  cities\_list.push\_back(i);  }  **for** (**size\_t** i = **0**; i < path\_len; i++)  swap(path[rand() % (path\_len - **1**) + **1**],  path[rand() % (path\_len - **1**) + **1**]);  **return** **0**;  }  **int** make\_area(**int** n\_cities)  {  **if** (get\_right\_path(n\_cities))  **return** -**1**;  local\_area.clear();  **for** (**int** i = **0**; i < n\_cities; i++)  local\_area.add\_city(i);  **for** (**int** i = **0**; i < (n\_cities - **1**); i++)  local\_area.add\_city\_road(path[i], path[i + **1**], **1**);  local\_area.add\_city\_road(**0**, n\_cities - **1**, **1**);  **for** (**int** i = **0**; i < n\_cities - **1**; i++)  **for** (**int** j = i + **1**; j < n\_cities; j++)  local\_area.add\_city\_road(i, j, (rand() % **10**) + **1**);  vector<**int**> cities\_list;  **for** (**int** i = **0**; i < n\_cities; i++)  cities\_list.push\_back(i);  **return** **0**;  }  pair<**double**, **int**> experiment(  **int** n\_cities,  **int** population\_size,  **int** n\_iterations,  **int** mutation\_percent)  {  srand(**0**);  make\_area(n\_cities);  vector<individual> population;  **double** time = -wtime();  population = find\_shortest\_path(  population\_size, cities\_list, n\_iterations, mutation\_percent);  time += wtime();  **int** result\_fitness = population[**0**].fitness();  **return** **make\_pair**(time, result\_fitness);  }  **void** make\_experiment(  **int** n\_cities,  **int** population\_size,  **int** mutation\_percent,  **int** n\_iterations,  **int** step,  options option,  string filename)  {  pair<**double**, **int**> result;  ofstream **file**(filename);  cout << filename << ":**\n**";  cout << left << setw(**20**) << "n\_cities" << setw(**20**) << "population\_size"  << setw(**20**) << "mutation\_percent" << setw(**20**) << "n\_iterations"  << setw(**20**) << "time" << setw(**20**) << "path\_len" << setw(**20**)  << "accuracy" << endl;  file << left << setw(**20**) << "n\_cities" << setw(**20**) << "population\_size"  << setw(**20**) << "mutation\_percent" << setw(**20**) << "n\_iterations"  << setw(**20**) << "time" << setw(**20**) << "path\_len" << setw(**20**)  << "accuracy" << endl;  **bool** experiment\_done = false;  **int** end\_value;  **switch** (option) {  **case** n\_cities\_opt:  end\_value = n\_cities;  n\_cities = step;  **break**;  **case** population\_size\_opt:  end\_value = population\_size;  population\_size = step;  **break**;  **case** mutation\_percent\_opt:  end\_value = mutation\_percent;  mutation\_percent = **0**;  **break**;  **case** n\_iterations\_opt:  end\_value = n\_iterations;  n\_iterations = step;  **break**;  **default:**  **return**;  }  **while** (!experiment\_done) {  result = experiment(  n\_cities, population\_size, n\_iterations, mutation\_percent);  file << left << setw(**20**) << n\_cities << setw(**20**) << population\_size  << setw(**20**) << mutation\_percent << setw(**20**) << n\_iterations  << setw(**20**) << result.first << setw(**20**) << result.second  << setw(**20**) << ((**double**)n\_cities / (**double**)result.second) \* **100**  << endl;  cout << left << setw(**20**) << n\_cities << setw(**20**) << population\_size  << setw(**20**) << mutation\_percent << setw(**20**) << n\_iterations  << setw(**20**) << result.first << setw(**20**) << result.second  << setw(**20**) << ((**double**)n\_cities / (**double**)result.second) \* **100**  << endl;  **switch** (option) {  **case** n\_cities\_opt:  n\_cities += step;  **if** (n\_cities > end\_value)  experiment\_done = true;  **break**;  **case** population\_size\_opt:  population\_size += step;  **if** (population\_size > end\_value)  experiment\_done = true;  **break**;  **case** mutation\_percent\_opt:  mutation\_percent += step;  **if** (mutation\_percent > end\_value)  experiment\_done = true;  **break**;  **case** n\_iterations\_opt:  n\_iterations += step;  **if** (n\_iterations > end\_value)  experiment\_done = true;  **break**;  }  }  file.close();  }  **int** main(**int** argc, **char**\* argv[])  {  **if** (argc < **2**)  **return** **1**;  **if** (strcmp(argv[**1**], "1") == **0**) {  make\_experiment(  **100**,  **4**,  **100**,  **10000**,  **10**,  n\_cities\_opt,  "/home/pahansan/genalg/data/n\_cities\_exp.csv");  make\_experiment(  **100**,  **100**,  **100**,  **10000**,  **5**,  population\_size\_opt,  "/home/pahansan/genalg/data/population\_size\_exp.csv");  make\_experiment(  **100**,  **4**,  **100**,  **10000**,  **1**,  mutation\_percent\_opt,  "/home/pahansan/genalg/data/mutation\_percent\_exp.csv");  make\_experiment(  **100**,  **4**,  **100**,  **50000**,  **1000**,  n\_iterations\_opt,  "/home/pahansan/genalg/data/n\_iterations\_exp.csv");  }  **if** (strcmp(argv[**1**], "2") == **0**) {  make\_area(**20**);  vector<individual> population  = find\_shortest\_path(**4**, cities\_list, **1000**, **100**);  srand(**0**);  **int** i = **0**;  **while** (population[**0**].fitness() != cities\_list.size()) {  population = find\_shortest\_path(**4**, cities\_list, **1000**, **100**);  i++;  }  cout << "fitness = " << population[**0**].fitness() << endl;  cout << i << endl;  }  **return** **0**;  } |

# Отзыв о работе студента

|  |
| --- |
|  |
| (ФИО студента) |
|  |

# Уровень освоения компетенций

|  |  |  |
| --- | --- | --- |
|  |  | |
|  | (ФИО студента) | |
|  |  | |
| Компетенции | | Уровень сформированности  компетенций |
| *ОПК-1 - Способен применять фундаментальные знания, полученные в области математических и (или) естественных наук, и использовать их в профессиональной деятельности* | |  |

отметка о зачете с оценкой \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Руководитель практики от СибГУТИ:

Заведующий кафедрой ВС Перышкова Евгения Николаевна

Должность руководителя подпись ФИО руководителя

"25" мая 2024 г.