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

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 v_i, $i = [1, 2, 3, \dots, n]$ взвешенного графа A, у которого каждая вершина соединена со всеми остальными при помощи рёбер e_k, $k = [1, 2, 3, \dots, \sum_{k=1}^{n-1} i]$ с весом $w_k \geq 1$. 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 g_j, соответствующее номеру i вершины v_i графа A;

- Геном – вектор $G=\{g_j\}$ - 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 $I=\{d_i\}$ 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 $0 \leq m \leq 100$, 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: $\text{точность} = \frac{\text{Количество городов}}{\text{Приспособленность первой особи}} \cdot 100\%$. 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).

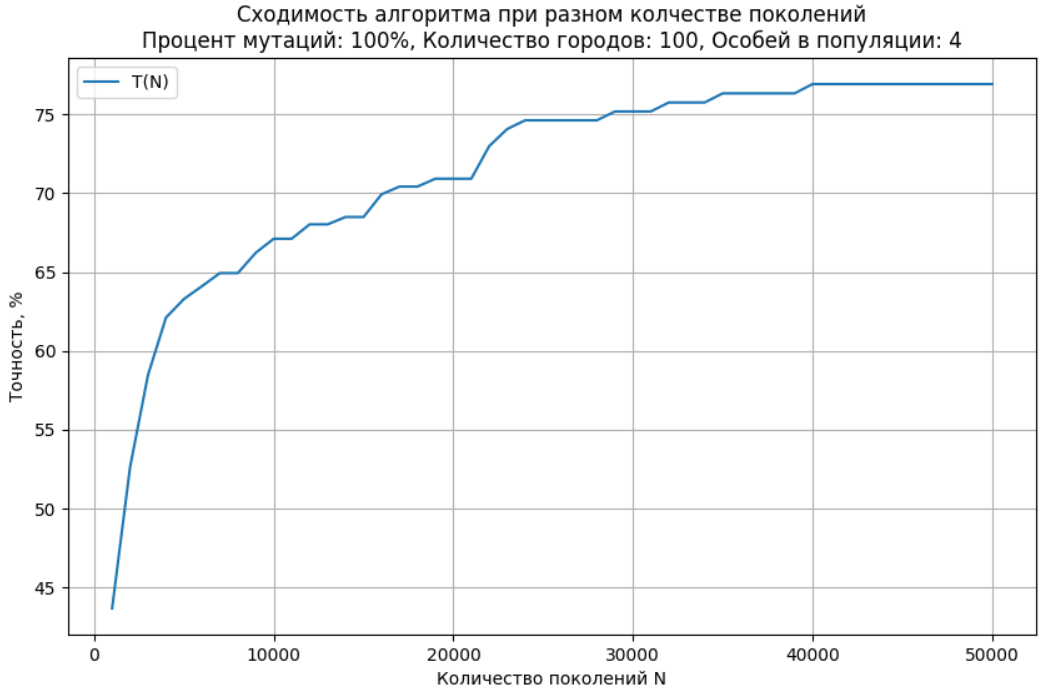

Рисунок 3. Сходимость алгоритма при разном количестве поколений.

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.

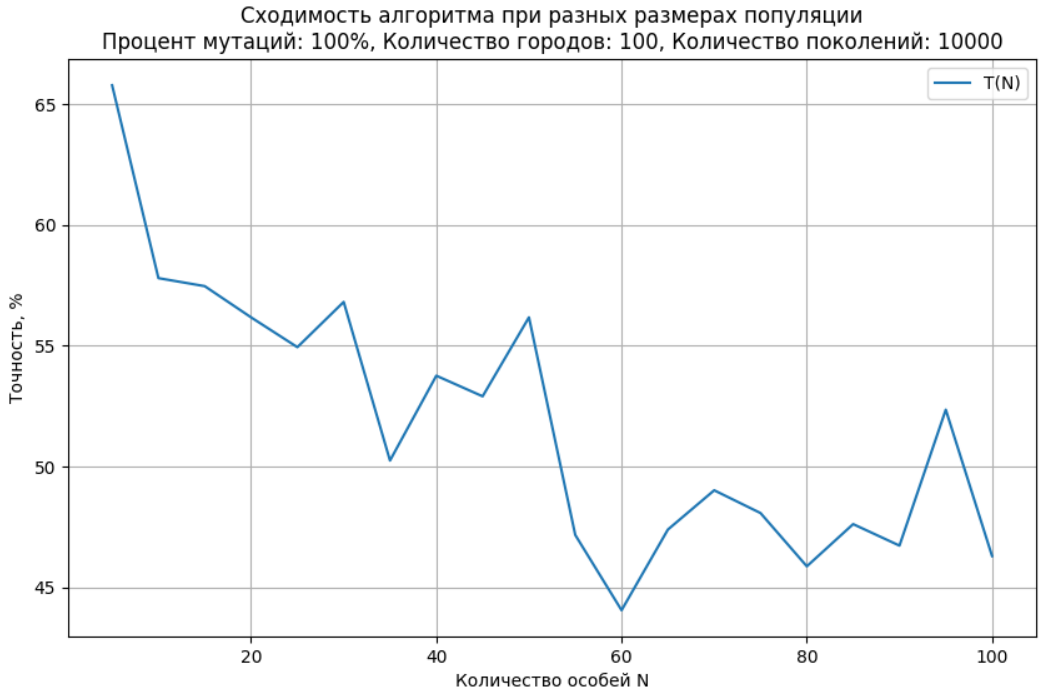

Рисунок 5. Сходимость алгоритма при разных размерах популяции.

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.

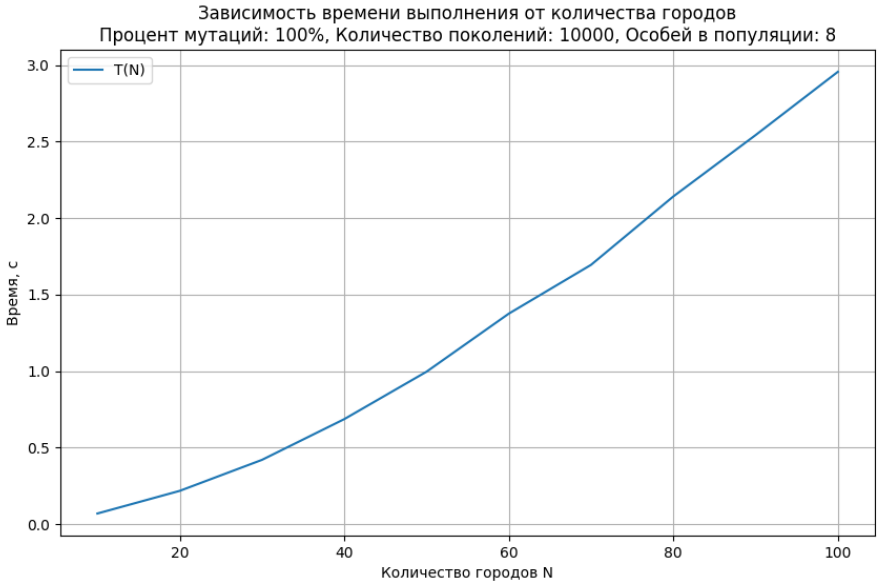

Рисунок 7. Зависимость времени выполнения от количества городов.

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).

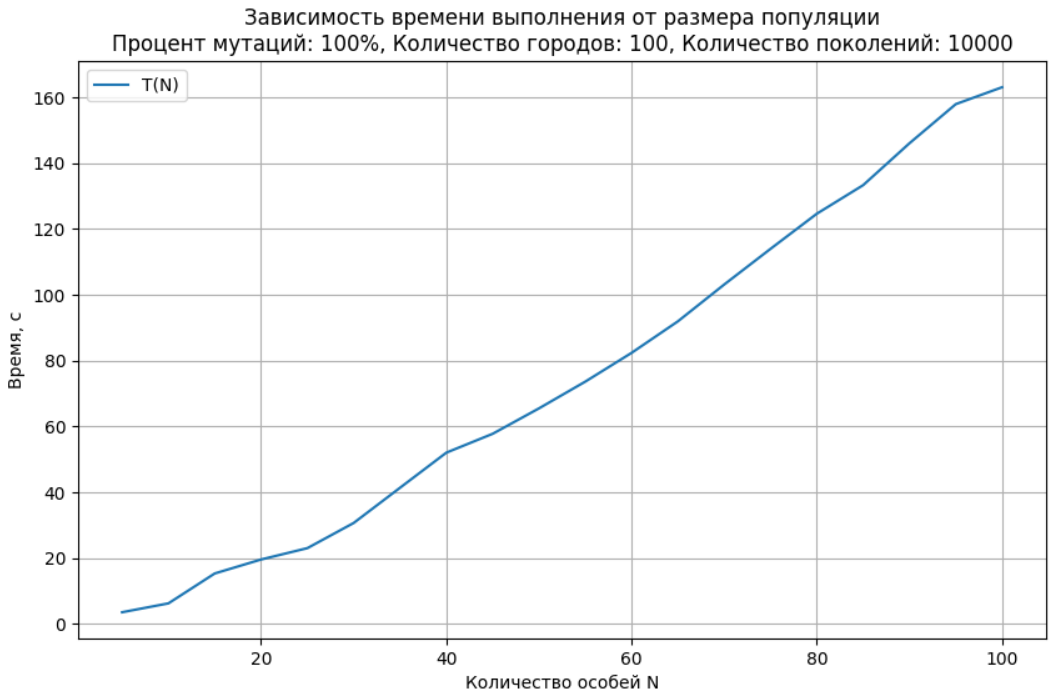

Рисунок 8. Зависимость времени выполнения от размера популяции.

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”

```
#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)
    {
```

```
54          name = new_name;
55      }
56      int get_name()
57      {
58          return name;
59      }
60      city& operator=(const city& other)
61      {
62          this->name = other.name;
63          this->roads = other.roads;
64          return *this;
65      }
66  };
67
68  class area {
69      vector<city> cities;
70
71  public:
72      area()
73      {
74      }
75      int add_city(int new_city_name)
76      {
77          city new_city(new_city_name);
78          for (auto& c : cities) {
79              if (c.get_name() == new_city_name)
80                  return -1;
81          }
82          cities.push_back(new_city);
83          return 0;
84      }
85      int add_city_road(int a, int b, int length)
86      {
87          if (a == b || length < 1)
88              return -1;
89          int a_exist = 0, b_exist = 0;
90          int a_city_name = a < b ? a : b;
91          int b_city_name = b > a ? b : a;
92          city a_city;
93          for (auto& c : cities) {
94              if (c.get_name() == a_city_name) {
95                  a_exist = 1;
96                  a_city = c;
97              }
98              if (c.get_name() == b_city_name) {
99                  b_exist = 1;
100             }
101         }
102         if (!a_exist || !b_exist)
103             return -1;
104         if (a_city.get_road(b_city_name) != -1)
105             return -1;
106         a_city.add_road(b_city_name, length);
107         for (auto& c : cities)
108             if (c.get_name() == a_city_name)
109                 c = a_city;
110         return 0;
111     }
```

```
112     int get_city_road(int a, int b)
113     {
114         if (a == b)
115             return -1;
116         int a_city_name = a < b ? a : b;
117         int b_city_name = b > a ? b : a;
118         for (auto& c : cities) {
119             if (c.get_name() == a_city_name)
120                 return c.get_road(b_city_name);
121         }
122         return -1;
123     }
124     void print()
125     {
126         for (auto& city_a : cities)
127             for (auto& city_b : cities) {
128                 if (city_a.get_name() == city_b.get_name())
129                     continue;
130                 cout << city_a.get_name() << " --> " << city_b.get_name()
131                      << " : "
132                      << get_city_road(city_a.get_name(), city_b.get_name())
133                      << endl;
134             }
135     }
136     void clear()
137     {
138         cities.clear();
139     }
140     int get_size()
141     {
142         return cities.size();
143     }
144 };
145
146 area local_area;
147 vector<int> path;
148 vector<int> cities_list;
149
150 class individual {
151     vector<int> genome;
152     int genome_size;
153
154 public:
155     individual() : genome_size(0)
156     {
157     }
158     individual(vector<int> new_genome, int size)
159         : genome(new_genome), genome_size(size)
160     {
161     }
162     int add_chromosome(int chromosome)
163     {
164         for (auto& g : genome)
165             if (g == chromosome)
166                 return -1;
167
168         genome.push_back(chromosome);
169         genome_size++;
```

```
170         return 0;
171     }
172     vector<int> get_genome()
173     {
174         return genome;
175     }
176     void make_mutation()
177     {
178         int rand_limit = genome_size - 1;
179         int a = rand() % rand_limit + 1;
180         int b = rand() % rand_limit + 1;
181         while (a == b)
182             b = rand() % rand_limit + 1;
183         swap(genome[a], genome[b]);
184     }
185     int size()
186     {
187         return genome_size;
188     }
189     size_t fitness() const
190     {
191         size_t length = 0;
192         int prev = -1;
193         int count = 0;
194         for (auto& chromosome : genome) {
195             count++;
196             if (prev == -1) {
197                 prev = chromosome;
198                 continue;
199             }
200             length += local_area.get_city_road(prev, chromosome);
201             prev = chromosome;
202             if (count == genome_size)
203                 length += local_area.get_city_road(0, chromosome);
204         }
205         return length;
206     }
207 };
208
209 bool operator<(const individual& a, const individual& b)
210 {
211     return a.fitness() < b.fitness();
212 }
213
214 void crossing(vector<individual>& population, int mutation_chance)
215 {
216     int population_size = population.size();
217
218     int a_number = rand() % population_size;
219     int b_number = rand() % population_size;
220
221     while (a_number == b_number)
222         b_number = rand() % population_size;
223
224     vector<int> genome_a = population[a_number].get_genome();
225     vector<int> genome_b = population[b_number].get_genome();
226
227     int genome_size = genome_a.size();
```

```
228     int cross_point = rand() % (genome_size - 1) + 1;
229
230     individual child_a;
231     individual child_b;
232     for (int i = 0; i < cross_point; i++) {
233         child_a.add_chromosome(genome_a[i]);
234         child_b.add_chromosome(genome_b[i]);
235     }
236     for (int i = cross_point; i < genome_size; i++) {
237         child_a.add_chromosome(genome_b[i]);
238         child_b.add_chromosome(genome_a[i]);
239     }
240     if (child_a.size() < genome_size) {
241         for (int i = cross_point; i < genome_size; i++) {
242             child_a.add_chromosome(genome_a[i]);
243             child_b.add_chromosome(genome_b[i]);
244         }
245     }
246
247     if (rand() % 100 < mutation_chance) {
248         child_a.make_mutation();
249         child_b.make_mutation();
250     }
251
252     population.push_back(child_a);
253     population.push_back(child_b);
254 }
255
256 void selection(vector<individual>& population, size_t population_size)
257 {
258     sort(population.begin(), population.end());
259     population.erase(population.begin() + population_size, population.end());
260 }
261
262 vector<individual>
263 make_initial_population(size_t population_size, vector<int> genome)
264 {
265     individual adam(genome, genome.size());
266     individual eva(genome, genome.size());
267
268     eva.make_mutation();
269
270     vector<individual> population = {adam, eva};
271
272     while (population.size() < population_size)
273         crossing(population, 100);
274
275     selection(population, population_size);
276
277     return population;
278 }
279
280 vector<individual> find_shortest_path(
281         int population_size,
282         vector<int> cities_list,
283         int n_iterations,
284         int mutation_percent)
285 {
```

```
286     vector<individual> population
287             = make_initial_population(population_size, cities_list);
288     for (int i = 0; i < n_iterations; i++) {
289         crossing(population, mutation_percent);
290         selection(population, population_size);
291     }
292     return population;
293 }
294
295 int get_right_path(size_t path_len)
296 {
297     if (path_len == path.size())
298         return -1;
299
300     path.clear();
301     cities_list.clear();
302
303     for (size_t i = 0; i < path_len; i++) {
304         path.push_back(i);
305         cities_list.push_back(i);
306     }
307     for (size_t i = 0; i < path_len; i++)
308         swap(path[rand() % (path_len - 1) + 1],
309              path[rand() % (path_len - 1) + 1]);
310
311     return 0;
312 }
313
314 int make_area(int n_cities)
315 {
316     if (get_right_path(n_cities))
317         return -1;
318
319     local_area.clear();
320
321     for (int i = 0; i < n_cities; i++)
322         local_area.add_city(i);
323
324     for (int i = 0; i < (n_cities - 1); i++)
325         local_area.add_city_road(path[i], path[i + 1], 1);
326
327     local_area.add_city_road(0, n_cities - 1, 1);
328
329     for (int i = 0; i < n_cities - 1; i++)
330         for (int j = i + 1; j < n_cities; j++)
331             local_area.add_city_road(i, j, (rand() % 10) + 1);
332
333     vector<int> cities_list;
334     for (int i = 0; i < n_cities; i++)
335         cities_list.push_back(i);
336
337     return 0;
338 }
339
340 pair<double, int> experiment(
341         int n_cities,
342         int population_size,
343         int n_iterations,
```

```
344         int mutation_percent)
345 {
346     srand(0);
347 
348     make_area(n_cities);
349 
350     vector<individual> population;
351     double time = -wtime();
352     population = find_shortest_path(
353             population_size, cities_list, n_iterations, mutation_percent);
354     time += wtime();
355     int result_fitness = population[0].fitness();
356     return make_pair(time, result_fitness);
357 }
358 
359 void make_experiment(
360         int n_cities,
361         int population_size,
362         int mutation_percent,
363         int n_iterations,
364         int step,
365         options option,
366         string filename)
367 {
368     pair<double, int> result;
369     ofstream file(filename);
370     cout << filename << ":\n";
371     cout << left << setw(20) << "n_cities" << setw(20) << "population_size"
372          << setw(20) << "mutation_percent" << setw(20) << "n_iterations"
373          << setw(20) << "time" << setw(20) << "path_len" << setw(20)
374          << "accuracy" << endl;
375     file << left << setw(20) << "n_cities" << setw(20) << "population_size"
376          << setw(20) << "mutation_percent" << setw(20) << "n_iterations"
377          << setw(20) << "time" << setw(20) << "path_len" << setw(20)
378          << "accuracy" << endl;
379     bool experiment_done = false;
380     int end_value;
381 
382     switch (option) {
383     case n_cities_opt:
384         end_value = n_cities;
385         n_cities = step;
386         break;
387     case population_size_opt:
388         end_value = population_size;
389         population_size = step;
390         break;
391     case mutation_percent_opt:
392         end_value = mutation_percent;
393         mutation_percent = 0;
394         break;
395     case n_iterations_opt:
396         end_value = n_iterations;
397         n_iterations = step;
398         break;
399     default:
400         return;
401     }
```

```
402
403     while (!experiment_done) {
404         result = experiment(
405                 n_cities, population_size, n_iterations, mutation_percent);
406         file << left << setw(20) << n_cities << setw(20) << population_size
407              << setw(20) << mutation_percent << setw(20) << n_iterations
408              << setw(20) << result.first << setw(20) << result.second
409              << setw(20) << ((double)n_cities / (double)result.second) * 100
410              << endl;
411         cout << left << setw(20) << n_cities << setw(20) << population_size
412              << setw(20) << mutation_percent << setw(20) << n_iterations
413              << setw(20) << result.first << setw(20) << result.second
414              << setw(20) << ((double)n_cities / (double)result.second) * 100
415              << endl;
416         switch (option) {
417         case n_cities_opt:
418             n_cities += step;
419             if (n_cities > end_value)
420                 experiment_done = true;
421             break;
422         case population_size_opt:
423             population_size += step;
424             if (population_size > end_value)
425                 experiment_done = true;
426             break;
427         case mutation_percent_opt:
428             mutation_percent += step;
429             if (mutation_percent > end_value)
430                 experiment_done = true;
431             break;
432         case n_iterations_opt:
433             n_iterations += step;
434             if (n_iterations > end_value)
435                 experiment_done = true;
436             break;
437         }
438     }
439     file.close();
440 }
441
442 int main(int argc, char* argv[])
443 {
444     if (argc < 2)
445         return 1;
446     if (strcmp(argv[1], "1") == 0) {
447         make_experiment(
448                 100,
449                 4,
450                 100,
451                 10000,
452                 10,
453                 n_cities_opt,
454                 "/home/pahansan/genalg/data/n_cities_exp.csv");
455         make_experiment(
456                 100,
457                 100,
458                 100,
459                 10000,
```

```
460                     5,
461                     population_size_opt,
462                     "/home/pahansan/genalg/data/population_size_exp.csv");
463             make_experiment(
464                     100,
465                     4,
466                     100,
467                     10000,
468                     1,
469                     mutation_percent_opt,
470                     "/home/pahansan/genalg/data/mutation_percent_exp.csv");
471             make_experiment(
472                     100,
473                     4,
474                     100,
475                     50000,
476                     1000,
477                     n_iterations_opt,
478                     "/home/pahansan/genalg/data/n_iterations_exp.csv");
479         }
480         if (strcmp(argv[1], "2") == 0) {
481             make_area(20);
482             vector<individual> population
483                     = find_shortest_path(4, cities_list, 1000, 100);
484             srand(0);
485             int i = 0;
486             while (population[0].fitness() != cities_list.size()) {
487                 population = find_shortest_path(4, cities_list, 1000, 100);
488                 i++;
489             }
490             cout << "fitness = " << population[0].fitness() << endl;
491             cout << i << endl;
492         }
493         return 0;
494 }
```

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.