

Санкт-Петербургский государственный университет

Прикладная математика, информатика и искусственный интеллект

Отчёт по учебной практике 2 (научно-исследовательской работе) (семестр 2)

«РЕАЛИЗАЦИЯ И ВИЗУАЛИЗАЦИЯ НЕКОТОРЫХ АЛГОРИТМОВ  
ЧИСЛЕННОЙ ОПТИМИЗАЦИИ»

Выполнил:


Дудко Тимофей Андреевич  
группа — 23.Б-05-мм



Научный руководитель:

к.ф.-м.н., доцент Шпилёв Пётр Валерьевич  
Кафедра Статистического моделирования

Работа выполнена  
в полном объеме



Отметка о зачёте:

Отметка о зачете:

<< Работа выполнена на хорошем уровне и может быть зачтена с оценкой С >>

Санкт-Петербург

2024

Отзыв на учебную практику 2 (научно – исследовательскую работу)  
студента 1-го курса бакалавриата Дудко Тимофея Андреевича

Работа посвящена изучению принципов математического моделирования. В рамках данной работы студент самостоятельно ознакомился с двумя алгоритмами численной оптимизации: алгоритмом имитации отжига и генетическим алгоритмом. Данные алгоритмы были реализованы студентом на языке Python, а их работа проиллюстрирована на примере задачи коммивояжера. Результаты тестирования показали, что для данной задачи алгоритм имитации отжига эффективнее.

Работа написана аккуратно, поставленная задача реализована в достаточно полном объеме. Считаю, что работа может быть зачтена с оценкой С.



16.05.24

Петр Валерьевич Шпилев

# Оглавление

|                                |           |
|--------------------------------|-----------|
| <b>Введение</b>                | <b>2</b>  |
| <b>Формулировка задачи</b>     | <b>3</b>  |
| <b>Генетический алгоритм</b>   | <b>3</b>  |
| Алгоритм . . . . .             | 3         |
| Реализация алгоритма . . . . . | 3         |
| Графики . . . . .              | 7         |
| Вывод по алгоритму . . . . .   | 9         |
| <b>Метод имитации отжига</b>   | <b>10</b> |
| Алгоритм . . . . .             | 10        |
| Реализация алгоритма . . . . . | 12        |
| Графики . . . . .              | 13        |
| Вывод по алгоритму . . . . .   | 14        |
| <b>Сравнение алгоритмов</b>    | <b>15</b> |
| <b>Заключение</b>              | <b>16</b> |
| <b>Список литературы</b>       | <b>17</b> |
| <b>Приложения</b>              | <b>18</b> |

## Введение

Оптимизация - это раздел вычислительной математики. Он занимается разработкой алгоритмов для поиска оптимального решения задачи, то есть нахождения минимума или максимума вещественной функции в некоторой области.

Свою актуальность он получил за счёт того, что прикладные задачи чаще всего не имеют аналитического решения, а даже если и имеют, то для его поиска потребуется слишком много времени и усилий. Но не стоит думать, что алгоритмы приходят к точному результату. В первую очередь они направлены на поиск наиболее желаемого результата, но не обязательного самого лучшего. Кроме того, существуют так называемые эвристические алгоритмы, то есть такие, правильность которых для всех возможных случаев не доказана.

Задачи оптимизации делятся на большое количество классов, от классов зависит выбор того или иного алгоритма. Это связано с тем, что функции могут иметь совершенно разный вид.

В этой работе я напишу код на языке программирования Python следующих алгоритмов: имитации отжига, генетический. С их помощью решу задачу коммивояжера.

## Формулировка задачи

Данные алгоритмы будут рассмотрены на задаче коммивояжера. Её можно представить в виде модели на графе. вершины графа соответствуют городам, а рёбра  $(i, j)$  между вершинами  $i$  и  $j$  - пути сообщения между этими городами. Каждому ребру  $(i, j)$  можно сопоставить критерий выгодности маршрута  $c_{ij} \geq 0$ , который можно понимать как, например, расстояние между городами. Считается, что модельный граф задачи является полностью связным.

Таким образом, решение задачи коммивояжёра — это нахождение гамильтонова цикла минимального веса в полном взвешенном графе, возвращающегося в начальную вершину. (Гамильтоновым циклом называется маршрут, включающий ровно по одному разу каждую вершину графа.).

## Генетический алгоритм

Генетический алгоритм - эвристический алгоритм, используемый для решения задач оптимизации путём случайного подбора и комбинирования с использованием механизмов, аналогичных естественному отбору в природе. Задача должна быть сформулирована так, чтобы решение могло быть закодировано в виде вектора генов.

### Алгоритм

#### Подготовка:

1. Задать целевую функцию для особей популяции
2. Создать начальную популяцию случайным образом

#### Начало цикла:

1. Скрещивание
2. Мутирование
3. Вычисление значения целевой функции для всех особей
4. Формирование нового поколения
5. Если выполняется условие остановки, то конец цикла, иначе начало цикла

### Реализация алгоритма

В качестве условия остановки алгоритма будет повторение лучших результатов четырех популяций подряд.

```

1 def generatePeaks(N, x_size=(-5, 5), y_size=(-5, 5)):
2     X = [random.uniform(x_size[0], x_size[1]) for _ in range(N)]
3     Y = [random.uniform(y_size[0], y_size[1]) for _ in range(N)]
4     return X, Y

```

### Функция **generatePeaks**

Генерирует вершины графа в виде пары точек (x, y). Возвращает X - все значения x и Y - все значения y

- N - количество вершин графа
- x\_size - кортеж, задающий ограничения по значениям x
- y\_size - кортеж, задающий ограничений по значениям y

### Функция **generateWays**

```

1 def generateWays(X, Y):
2     N = len(X)    ways = dict()
3     for i in range(N):
4         for j in range(i+1, N):
5             x = ((X[i] - X[j])**2 + (Y[i]-Y[j])**2) ** 0.5
6             ways[i, j] = round(x, 3)
7     return ways

```

Возвращает словарь, ключами которого являются вершины графа, а значениями соответствующие веса ребер между ними. Вес ребра - евклидово расстояние между точками на плоскости.

- X - массив значений x
- Y - массив значений y

### Функция **FitnessMax**

```

1 def FitnessMax(ind, ways):
2     sum = 0
3     for i in range(len(ind)-1):
4         try:
5             sum += ways[ind[i], ind[i+1]]
6         except:
7             sum += ways[ind[i+1], ind[i]]
8     try:
9         sum += ways[ind[0], ind[len(ind)-1]]
10    except:
11        sum += ways[ind[len(ind) - 1], ind[0]]
12    return sum

```

Возвращает значение фитнес-функции особи

- ind - особь
- ways - пути графа

### Функция **Individual**

```

1 def Individual(N):
2     way = [0]
3     while len(way) != N:
4         x = random.randint(1, N - 1)
5         if x not in way:
6             way.append(x)
7     return way

```

Возвращает особь

- N - количество вершин графа

### Функция **Crossing**

```

1 def Crossing(parent1, parent2, ways):
2     N = len(parent1)
3     descendant = [-1] * N
4     slice = random.randint(1, N-1)
5     descendant[slice] = parent1[slice]
6     for i in range(slice, N):
7         if parent2[i] not in descendant:
8             descendant[i] = parent2[i]
9     if -1 not in descendant:
10        return (descendant, FitnessMax(descendant, ways))
11    else:
12        for j in range(slice, N):
13            if parent1[j] not in descendant:
14                descendant[descendant.index(-1)] = parent1[j]
15    return (descendant, FitnessMax(descendant, ways))

```

Скрещивает пару особей. Функция возвращает кортеж из потомка и его значения фитнесс-функции.

- parent1 - первый родитель
- parent2 - второй родитель
- ways - пути графа

### Функция **createPopulation**

Создает из особей популяцию. Возвращает список из всех особей и соответствующих им значений фитнесс-функции в порядке убывания.

```

1 def createPopulation(ways, count, N):
2     Population = []
3     for i in range(count):
4         Ind = Individual(N)
5         Population.append((Ind, FitnessMax(Ind, ways)))
6     Population = sorted(Population, key=lambda x: x[1], reverse=False)
7     return Population

```

- ways - пути графа
- count - количество особей в популяции
- N - количество вершин графа

### Функция **Mutant**

```

1 def Mutant(ways, individual):
2     N = len(individual)
3     p1 = random.randint(1, N - 1)
4     p2 = random.randint(1, N - 1)
5     new_individual = []
6     if p2 < p1:
7         for i in range(p2):
8             new_individual.append(individual[i])
9         new_individual += individual[p2:p1 + 1][::-1]
10        for i in range(p1 + 1, N):
11            new_individual.append(individual[i])
12    else:
13        for i in range(p1):
14            new_individual.append(individual[i])
15        new_individual += individual[p1:p2 + 1][::-1]
16        for i in range(p2 + 1, N):
17            new_individual.append(individual[i])
18    return (new_individual, FitnessMax(new_individual, ways))

```

Возвращает мутированную особь и ее значение фитнес-функции. Мутация происходит следующим путём: выбираются 2 случайные вершины и путь между ними инвертируется.

- ways - пути графа
- individual - особь



## Генетический алгоритм

```

1 def genetic(ways, count, chance, N):
2     Population = createPopulation(ways, count, N)
3     history = [] #история точности
4     for i in range(1000):
5         #Производим скрещивание и мутации
6         posterity = []
7         for p in range(count):
8             for j in range(p+1, count - 1):
9                 new_gen = Crossing(Population[p][0], Population[j][0], ways)
10                x = random.random() #генерируем случайное число, чтобы определить,
будет ли мутация
11                if x <= chance:
12                    new_gen = Mutant(ways, new_gen[0])
13                    posterity.append(new_gen)
14                Population = sorted(posterity, key=lambda x: x[1], reverse=False)
15                Population = Population[:count]
16                #Сохраняем точность лучшей особи
17                history.append(Population[0][1])
18                #Критерий остановки
19                if len(history) >= 4:
20                    if (history[-1] == history[-2] == history[-3] == history[-4]):
21                        break
22    return history[-1]

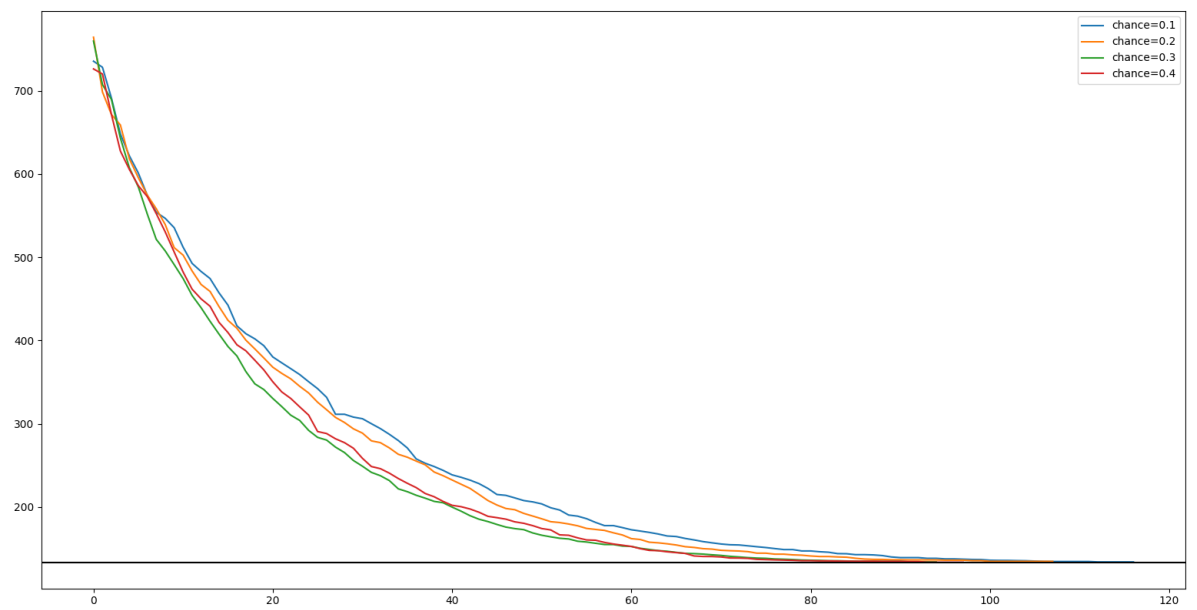
```

Возвращает наилучший результат

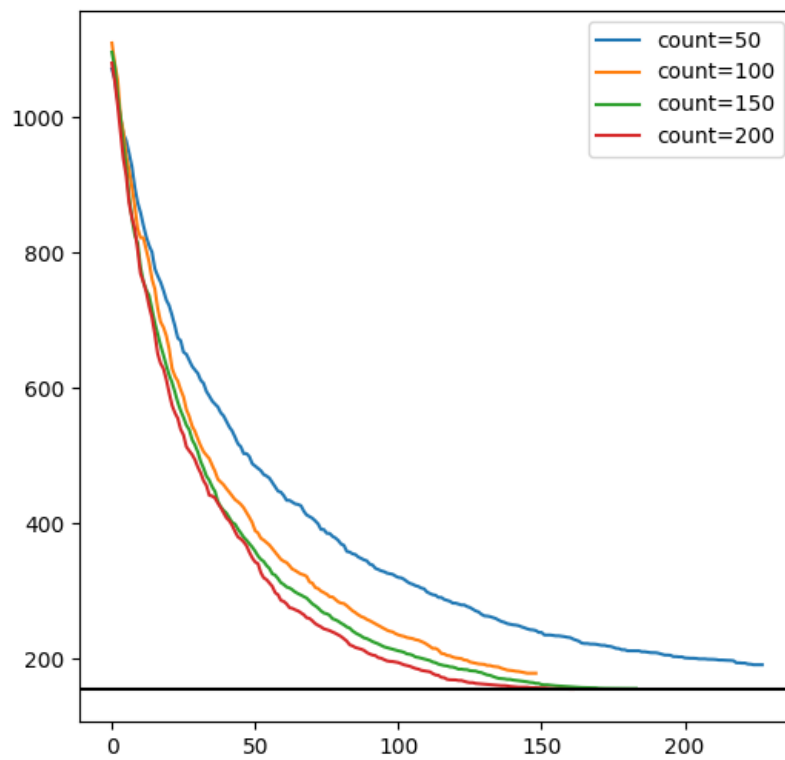
- ways - пути графа
- count - количество особей в популяции
- chance - шанс мутирования
- N - количество вершин графа

## Графики

Сходимость алгоритма при разных шансах мутации. Количество городов - 100, количество особей в популяции - 150.

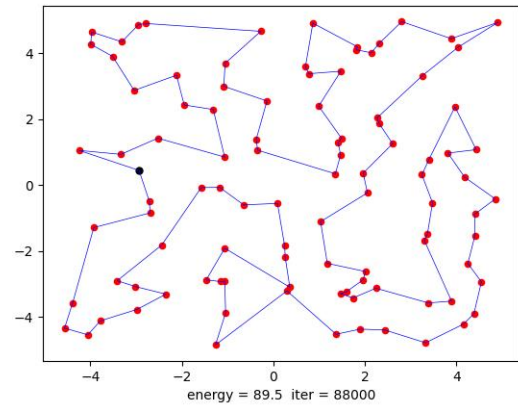
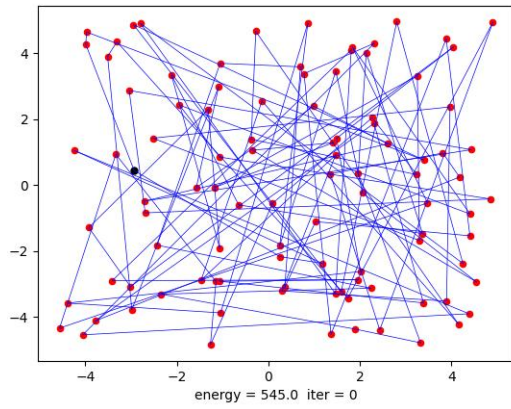


Сходимость алгоритма при разном количестве особей в популяции. Количество городов - 150, шанс мутации - 0.2



## Начальное и конечное решение

$N = 100$



## Вывод по алгоритму

Алгоритм показал хорошую работу и справился с задачей коммивояжера.

## Метод имитации отжига

Алгоритм основывается на имитации физического процесса, который происходит при кристаллизации вещества, в том числе при отжиге металлов. Предполагается, что атомы вещества уже почти выстроены в кристаллическую решётку, но ещё допустимы переходы отдельных атомов из одной ячейки в другую. Активность атомов тем больше, чем выше температура, которую постепенно понижают, что приводит к тому, что вероятность переходов в состояния с большей энергией уменьшается. Устойчивая кристаллическая решётка соответствует минимуму энергии атомов, поэтому атом либо переходит в состояние с меньшим уровнем энергии, либо остаётся на месте.

### Алгоритм

Моделирование похожего процесса используется для решения задачи глобальной оптимизации, состоящей в нахождении такой точки или множества точек, на которых достигается минимум некоторой целевой функции  $F(x)$  (энергия сестемы), где  $x \in X$  ( $x$  - состояние системы,  $X$  - множество всех состояний)

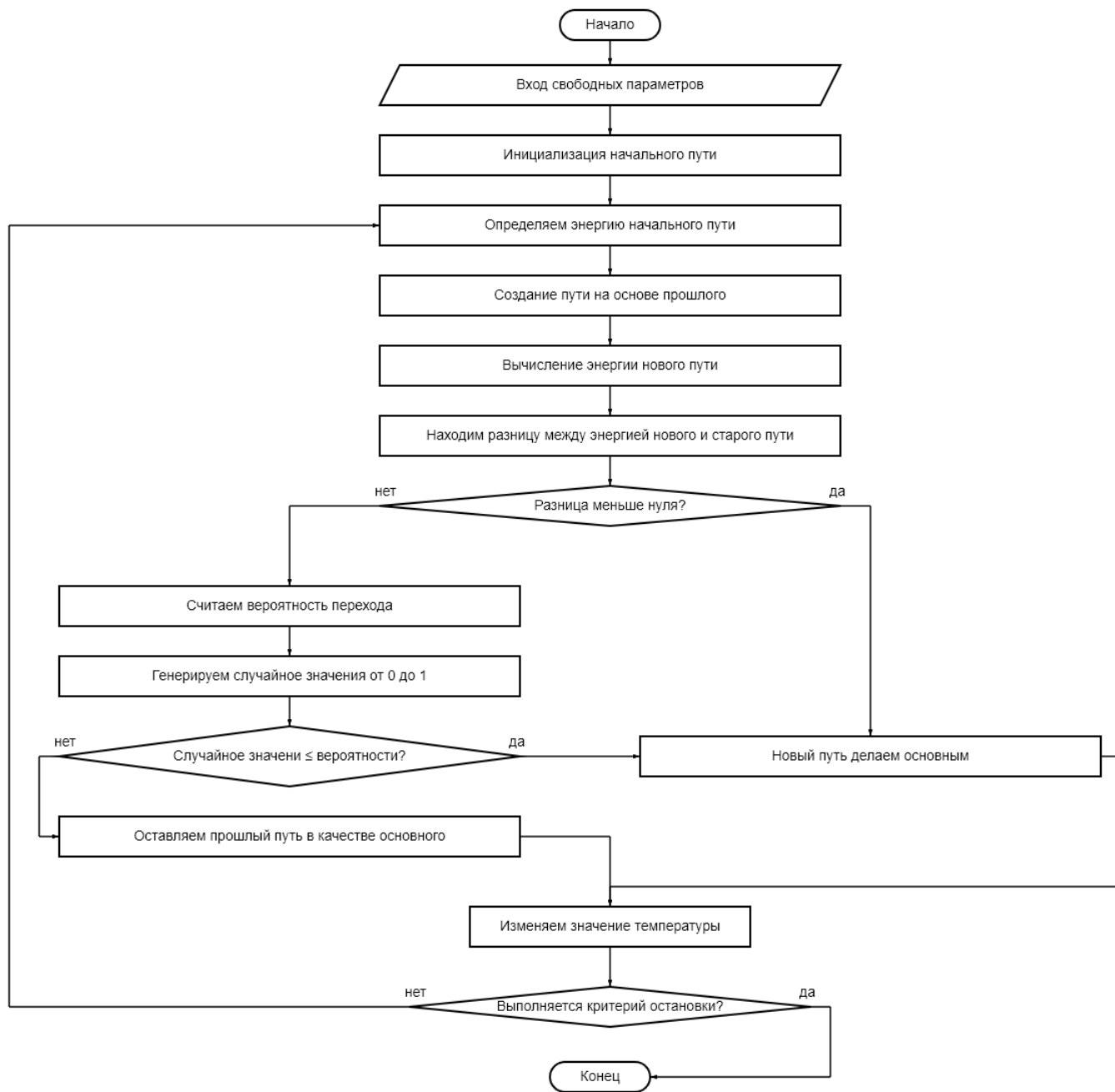
- $x_0 \in X$  - начальное состояние системы
- Оператор  $A(x, i) : X \times \mathbb{N} \rightarrow X$  случайно генерирует новое состояние системы после  $i$ -го шага с учетом текущего состояния  $x$
- $T_i > 0$  - убывающая к нулю положительная последовательность

Алгоритм генерирует процесс случайного блуждания по пространству состояний  $X$ . Решение ищется последовательным вычислением точек  $x_0, x_1, \dots$  пространства  $X$ . На каждом шаге алгоритм (который описан ниже) вычисляет новую точку и понижает значение величины (изначально положительной), понимаемой как "температура".

Последовательность этих точек получается следующим образом. К точке  $x_i$  применяется оператор  $A$ , в результате чего получается новое состояние  $x_i^* = A(x_i, i)$ , для которого вычисляется значение энергии.

$\Delta F_i = F(x_i^*) - F(x_i)$ . Если  $\Delta F_i \leq 0$ , тогда осуществляется переход  $x_{i+1} = x_i^*$ . Иначе переход осуществляется с некоторой вероятностью  $P(x_i^* \rightarrow x_{i+1} | x_i) = e^{\frac{-\Delta F_i}{T_i}}$ . Если переход не произошёл, то  $x_{i+1} = x_i$ . Алгоритм останавливается, когда  $T$  принимает заданное значение.

## Блок-схема алгоритма



## Реализация алгоритма

В качестве условия останова алгоритма будет достижение заданного значения температуры или повторение энергии 4 итерации подряд.

Функции **generatePeaks**, **generateWays**, **createPopulation** такие же. Функция нового состояния повторяет функцию **Mutant**. Функция начального состояния **randomState** повторяет функцию **Individual**. Функция **energy** повторяет функцию **FitnessMax**.

Функция **gibbs**

```
1 def gibbs(T, dE):
2     return np.exp(-dE/T)
```

Возвращает шанс перехода к следующему состоянию

- T - значение температуры
- dE - разница между энергией нового и прошлого состояний

Функция **changeT**

```
1 def changeT(start_T, C, iteration):
2     return start_T * C / iteration
```

Возвращает новое значение температуры

- start\_T - начальная температура
- C - коэффициент
- iteration - номер итерации

## Функция Метод имитации отжига

```

1 def annealing(start_T, N, C, eps):
2     history = []
3     iteration = 1
4     T = changeT(start_T, C, iteration)
5     X, Y = generatePeaks(N)
6     ways = generateWays(X, Y)
7     way0 = randomState(N)
8     while T >= eps:
9         way1 = newState(way0)
10        en0 = energy(way0, ways)
11        en1 = energy(way1, ways)
12        dE = en1 - en0
13        if dE <= 0:
14            way0 = way1
15        else:
16            p = gibbs(T, dE)
17            r = random.random()
18            if r <= p:
19                way0 = way1
20            iteration += 1
21            T = changeT(start_T, C, iteration)
22            history.append(en1)
23    return history[-1]
```

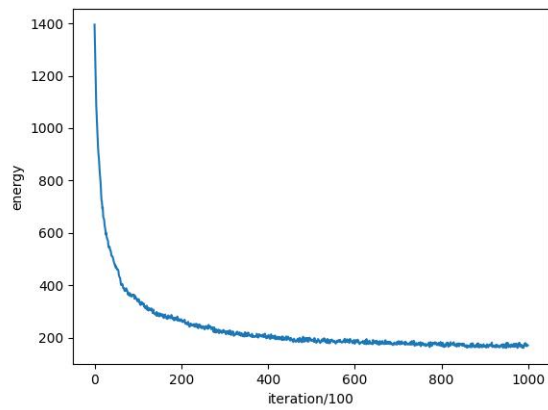
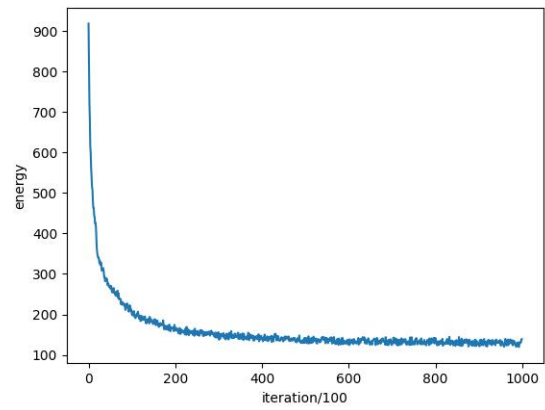
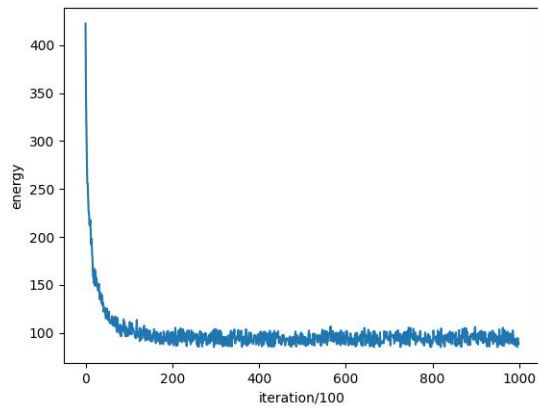
Возвращает оптимальную длину пути.

- start\_T - начальная температура
- N - количество вершины
- C - коэффициент для изменения температуры
- eps -  $T < \text{eps}$  для остановки алгоритма

## Графики

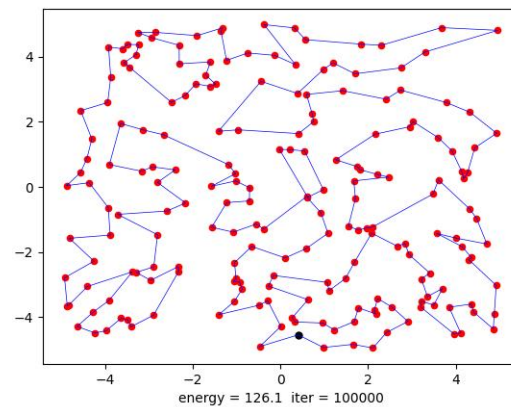
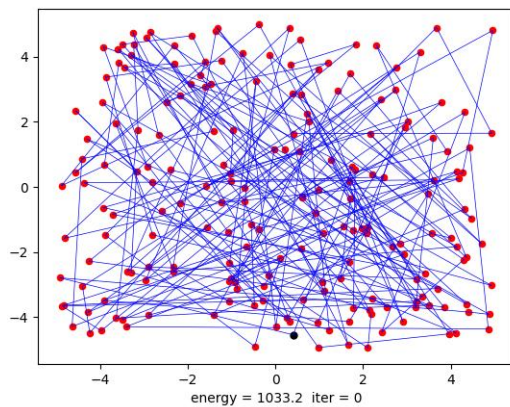
start\_T = 10, C = 0.1, eps = 0.00001

История энергии для 100/200 городов в зависимости от итерации



Начальное и конечное решения

$N = 200$



Вывод по алгоритму

Алгоритм смог хорошо решить задачу коммивояжера.



## Сравнение алгоритмов

Запуски алгоритмов при разных N.

### 1. Генетический алгоритм

- count = 150
- chance = 0.2

### 2. Имитация отжига

- start\_T = 10
- C = 0.1
- eps = 0.0001

| N   | Генетический |      |        | Имитация отжига |        |        |
|-----|--------------|------|--------|-----------------|--------|--------|
|     | time/s       | iter | res    | time/s          | iter   | res    |
| 50  | 6,53         | 46   | 62,71  | 2,61            | 100001 | 62,71  |
| 100 | 50,81        | 133  | 94,72  | 4,82            | 100001 | 94,72  |
| 150 | 124,83       | 168  | 119,03 | 6,99            | 100001 | 111,49 |
| 200 | 287,63       | 250  | 140,03 | 9,26            | 100001 | 132,79 |
| 250 | 602,94       | 366  | 157,43 | 11,80           | 100001 | 146,79 |
| 300 | 980,91       | 422  | 185,24 | 15,31           | 100001 | 167,38 |
| 350 | 1458,34      | 542  | 201,42 | 18,29           | 100001 | 189,83 |

## Заключение

В данной работе были изучены, реализованы и визуализированы 2 метода оптимизации - генетический алгоритм и метод имитации отжига. Алгоритмы запускались при разных гиперпараметрах. При их помощи была решена задача коммивояжера. По результатам запусков видно, что метод имитации отжига справляется с ней более эффективно. Для хорошего решения генетическим алгоритмом нужно большое количество особей в популяции, что сильно его замедляет.

## Список литературы

- [1] Генетический алгоритм (Wikipedia). [https://ru.wikipedia.org/wiki/Генетический\\_алгоритм](https://ru.wikipedia.org/wiki/Генетический_алгоритм).  
Обращение 01.04.2024.
- [2] Алгоритм имитации отжига (Wikipedia). [https://ru.wikipedia.org/wiki/Алгоритм\\_имитации\\_отж](https://ru.wikipedia.org/wiki/Алгоритм_имитации_отж).  
Обращение 20.04.2024.
- [3] А. П. Карпенко - "Современные алгоритмы поисковой оптимизации". Издательство «МГТУ им. Н. Э. Баумана», 2017г.
- [4] Лесин В.В., Лисовец Ю.П. - "Основы методов оптимизации". Издательство «Лань», 2022г.

## Приложения

### Прил. 1

```

1  import random
2  import matplotlib.pyplot as plt
3  import numpy as np
4
5  #Генерация пунктов
6  def generatePeaks(N, x_size=(-5, 5), y_size=(-5, 5)):
7      X = [random.uniform(x_size[0], x_size[1]) for _ in range(N)]
8      Y = [random.uniform(y_size[0], y_size[1]) for _ in range(N)]
9      return X, Y
10
11  #Расстояния между всеми пунктами
12  def generateWays(X, Y):
13      N = len(X)
14      ways = dict()
15      for i in range(N):
16          for j in range(i+1, N):
17              x = ((X[i] - X[j])**2 + (Y[i]-Y[j])**2) ** 0.5
18              ways[i, j] = round(x, 3)
19      return ways
20
21
22  #Функция энергии состояния
23  def energy(way, ways):
24      sum = 0
25      for i in range(len(way)-1):
26          try:
27              sum += ways[way[i], way[i+1]]
28          except:
29              sum += ways[way[i+1], way[i]]
30      try:
31          sum += ways[way[0], way[len(way)-1]]
32      except:
33          sum += ways[way[len(way) - 1], way[0]]
34      return sum
35

```

## Прил. 2

```

36 #Создает начальное состояние
37 def randomState(N):
38     way = [0]
39     while len(way) != N:
40         x = random.randint(1, N - 1)
41         if x not in way:
42             way.append(x)
43     return way
44
45 #Создает новое состояние на основе прошлого
46 def newState(way):
47     p1 = random.randint(1, len(way)-1)
48     p2 = random.randint(1, len(way)-1)
49
50     new_way = []
51
52     if p2 < p1:
53         for i in range(p2):
54             new_way.append(way[i])
55         new_way += way[p2:p1+1][::-1]
56         for i in range(p1+1, len(way)):
57             new_way.append(way[i])
58     else:
59         for i in range(p1):
60             new_way.append(way[i])
61         new_way += way[p1:p2+1][::-1]
62         for i in range(p2+1, len(way)):
63             new_way.append(way[i])
64
65     return new_way
66
67 #Вероятность перехода в новое состояние
68 def gibbs(T, dE):
69     return np.exp(-dE/T)

```

## Прил. 3

```

71 def changeT(start_T, C, iteration):
72     return start_T * C / iteration
73
74 #Алгоритм
75 def annealing(start_T, N, C, eps):
76     history = []
77     iteration = 1
78     T = changeT(start_T, C, iteration)
79     X, Y = generatePeaks(N)
80     ways = generateWays(X, Y)
81     way0 = randomState(N)
82     while T >= eps:
83         way1 = newState(way0)
84         en0 = energy(way0, ways)
85         en1 = energy(way1, ways)
86         dE = en1 - en0
87
88         if dE <= 0:
89             way0 = way1
90             history.append(en1)
91         else:
92             p = gibbs(T, dE)
93             r = random.random()
94             if r <= p:
95                 way0 = way1
96                 history.append(en1)
97
98         iteration += 1
99         T = changeT(start_T, C, iteration)
100         if len(history) >= 4:
101             if (history[-1] == history[-2] == history[-3] == history[-4]):
102                 break
103     return history[-1]

```

## Прил. 4

```

106 #Создает начальное состояние
107 def Individual(N):
108     way = [0]
109     while len(way) != N:
110         x = random.randint(1, N - 1)
111         if x not in way:
112             way.append(x)
113     return way
114
115 #Фитнесс-функция
116 def FitnessMax(ind, ways):
117     sum = 0
118     for i in range(len(ind)-1):
119         try:
120             sum += ways[ind[i], ind[i+1]]
121         except:
122             sum += ways[ind[i+1], ind[i]]
123     try:
124         sum += ways[ind[0], ind[len(ind)-1]]
125     except:
126         sum += ways[ind[len(ind) - 1], ind[0]]
127     return sum
128
129 #Создание популяции из count особей
130 def createPopulation(ways, count, N):
131     Population = []
132     for i in range(count):
133         Ind = Individual(N)
134         Population.append((Ind, FitnessMax(Ind, ways)))
135     Population = sorted(Population, key=lambda x: x[1], reverse=False)
136     return Population

```

## Прил. 5

```

138 #Мутация
139 def Mutant(ways, individual):
140     N = len(individual)
141     p1 = random.randint(1, N - 1)
142     p2 = random.randint(1, N - 1)
143     new_individual = []
144     if p2 < p1:
145         for i in range(p2):
146             new_individual.append(individual[i])
147         new_individual += individual[p2:p1 + 1][::-1]
148         for i in range(p1 + 1, N):
149             new_individual.append(individual[i])
150     else:
151         for i in range(p1):
152             new_individual.append(individual[i])
153         new_individual += individual[p1:p2 + 1][::-1]
154         for i in range(p2 + 1, N):
155             new_individual.append(individual[i])
156
157     return (new_individual, FitnessMax(new_individual, ways))
158
159 #Скрещивание
160 def Crossing(parent1, parent2, ways):
161     N = len(parent1)
162     descendant = [-1] * N
163     slice = random.randint(1, N-1)
164     descendant[:slice] = parent1[:slice]
165     for i in range(slice, N):
166         if parent2[i] not in descendant:
167             descendant[i] = parent2[i]
168     if -1 not in descendant:
169         return (descendant, FitnessMax(descendant, ways))
170     else:
171         for j in range(slice, N):
172             if parent1[j] not in descendant:
173                 descendant[descendant.index(-1)] = parent1[j]
174     return (descendant, FitnessMax(descendant, ways))

```



```

177 # Алгоритм
178 def genetic(ways, count, chance, N):
179     iterations = 0
180     Population = createPopulation(ways, count, N)
181     history = [] #история точности
182     for i in range(1000):
183         iterations += 1
184
185         #Производим скрещивание и мутации
186         posterity = []
187         for p in range(count):
188             for j in range(p+1, count - 1):
189                 new_gen = Crossing(Population[p][0], Population[j][0], ways)
190                 x = random.random() #генерируем случайное число, чтобы определить, будет ли мутация
191                 if x <= chance:
192                     new_gen = Mutant(ways, new_gen[0])
193                 posterity.append(new_gen)
194
195         Population = sorted(posterity, key=lambda x: x[1], reverse=False)
196         Population = Population[:count]
197
198         #Сохраняем точность лучшей особи
199         history.append(Population[0][1])
200         #plotGraph(X, Y, Population[0][0], iterations)
201
202         #Критерий остановки
203         if len(history) >= 4:
204             if (history[-1] == history[-2] == history[-3] == history[-4]):
205                 break
206
207     return history[-1]

```