

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
Национальный исследовательский университет
«Высшая школа Экономики»

ФАКУЛЬТЕТ ЭКОНОМИЧЕСКИХ НАУК

ОБРАЗОВАТЕЛЬНАЯ ПРОГРАММА «ЭКОНОМИКА»

КУРСОВАЯ РАБОТА

Стохастические методы оптимизации

Выполнил:
студент группы БЭК1812
Хайкин ГЛЕБ АЛЕКСЕЕВИЧ

Научный руководитель:
старший преподаватель
Борзых Дмитрий Александрович

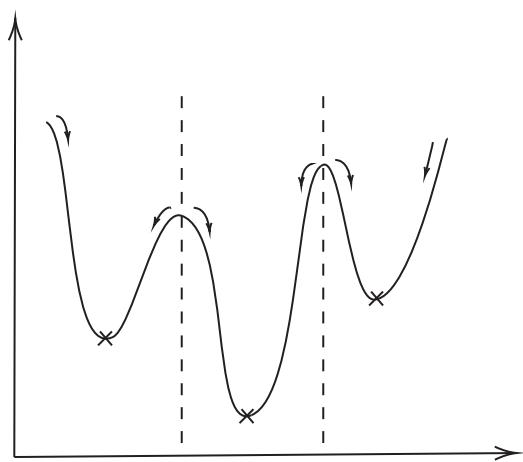


МОСКВА — 2020

Оглавление

Введение	3
1. Имитация отжига	4
1.1. Алгоритм	4
1.2. N ферзей	5
1.3. Минимизация негладкой функции	10
1.4. Задача коммивояжера	12
1.5. Вывод	15
2. Метод роения частиц	16
2.1. Алгоритм	16
2.2. Функция Розенброка	18
2.3. Вывод	23
3. Генетический алгоритм	25
3.1. Алгоритм	25
3.2. Функция Экли	28
3.3. Задача коммивояжера	34
3.4. Вывод	40
Заключение	41
Список литературы	42

Введение



1. Имитация отжига

Имитация отжига (*Simulated Annealing*, SA) представляет собой алгоритм решения задачи поиску глобального оптимума некоторой функции $F: \mathbb{X} \rightarrow \mathbb{R}$ через упорядоченный стохастический поиск, базирующийся на моделировании физического процесса кристаллизации вещества из жидкого состояния в твердое.

ДОБАВИТЬ ВВЕДНИЕ

1.1. Алгоритм

Для описания метода рассмотрим задачу нахождения глобального минимума:

$$F(x) \rightarrow \min_{x \in \mathbb{X}},$$

где $x = (x_1, \dots, x_m)$ — вектор всех состояний, \mathbb{X} — множество всех состояний.

Положим, что $k = 0$ и изначально температура зафиксированна на определенном уровне $T(k) = \text{const.}$

1. Из множества всех состояний выберем случайный элемент $\hat{x}(k) \equiv x_i$,
 $i \in (1, \dots, m).$
2. Понизим температуру одним из следующих способов:
 - a) Больцмановский отжиг

$$T(k) = \frac{T(0)}{\ln(1 + k)}, \quad k > 0 \quad (1.1)$$

- б) Отжиг Коши

$$T(k) = \frac{T(0)}{k} \quad (1.2)$$

- в) Метод тушения

$$T(k + 1) = \alpha T(k), \quad \alpha \in (0, 1) \quad (1.3)$$

3. Пусть следующий элемент зависит от функции из семейства симметричных вероятностных распределений $G: \mathbb{X} \rightarrow \mathbb{X}$, порождающей новое состояние:

$$\tilde{x}(k) \sim G(\hat{x}(k), T(k)).$$

- а) Часто G выбирается из семейства нормальных распределений:

$$G(\tilde{x}; \hat{x}, T) = \frac{1}{\sqrt{(2\pi)^D T}} \exp \left\{ \frac{-|\tilde{x} - \hat{x}|^2}{2T} \right\}, \quad (1.4)$$

где \hat{x} — математическое ожидание, T — дисперсия, D — размерность пространства всех состояний.

б) Также для $D = 1$ используется распределение Коши с плотностью:

$$G(\tilde{x}; \hat{x}, T) = \frac{1}{\pi} \frac{T}{|\tilde{x} - \hat{x}|^2 + T^2}, \quad (1.5)$$

где \hat{x} — параметр сдвига, T — параметр масштаба.

4. Рассчитываем разницу двух функций:

$$\Delta F = F(\tilde{x}(k)) - F(\hat{x}(k)).$$

5. Принимаем $\tilde{x}(k)$ за новый элемент, то есть $\hat{x}(k+1) \equiv \tilde{x}(k)$, с вероятностью

$$\mathbb{P}(\{\hat{x}(k+1) = \tilde{x}(k)\}) = \begin{cases} 1, & \Delta F < 0 \\ \exp\left\{-\frac{\Delta F}{T(k)}\right\}, & \Delta F \geq 0 \end{cases} \quad (1.6)$$

и отвергаем его, то есть $\hat{x}(k+1) \equiv \hat{x}(k)$, с вероятностью

$$q = 1 - \mathbb{P}(\{\hat{x}(k+1) = \tilde{x}(k)\}).$$

Заметим, чем выше температура, тем больше вероятность принять состояние хуже текущего ($\Delta F \geq 0$).

6. Возвращаемся к пункту 2, пока не достигнем глобального минимума.

1.2. N ферзей

Рассмотрим задачу, в которой необходимо расставить N ферзей на шахматной доске размера $N \times N$ так, чтобы ни один из них не «бил» другого.

В таком случае, множество всех состояний \mathbb{X} будет содержать всевозможные расстановки ферзей на шахматной доске. Общее число возможных расположений n ферзей на $N \times N$ -клеточной доске равно:

$$\binom{N \times N}{n} = \frac{N \times N!}{n!(N \times N - n)!}$$

Тогда функция $F : \mathbb{X} \rightarrow \mathbb{R}$ будет выдавать количество атак ферзей, и решением данной задачи будет нахождение такого расположения x^* , что $F(x^*) \equiv 0$.

Зафиксируем изначальное расположение ферзей на шахматной доске. Очевидно, что несколько ферзей не могут находиться на одной вертикали или горизонтали, ибо тогда они будут находиться под ударом друг-друга. Следовательно, наша задача сужается к поиску расположения:

$$x^* = (q_1, \dots, q_n) = \{(1, h_1), \dots, (n, h_n)\}, h_1 \neq \dots \neq h_n, \quad (1.7)$$

где (i, h_i) — расположение ферзя q_i на i -ой вертикали по горизонтали h_i . Отметим, что такая задача имеет $N!$ решений.

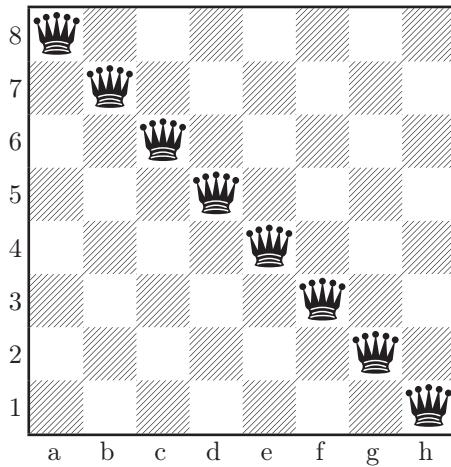


Рисунок 1.1 — Изначальное расположение.

Определим функцию, которая будет создавать изначальное неоптимальное расположение, в общем виде. Учтем, что несколько ферзей не могут находиться на одной вертикали или горизонтали.

```
In: def queens(N):
    ver = np.arange(1, N + 1)
    hor = np.arange(1, N + 1)
    np.random.shuffle(hor)
    return np.column_stack((ver, hor)) # получаем массив
# размерности (N, 2), отображающий расположение ферзей
```

Выведем первоначальное расположение ферзей для стандартной доски 8×8 , где первый столбец массива — расположение по вертикали, второй столбец массива — расположение по горизонтали. Для наглядности — презентации оптимизационного процесса — выстроим изначальную расстановку на главной диагонали (рис. 1.1).

```
In: matrix = queens(8)
matrix
```

```
Out: array([[1, 1],
           [2, 2],
           [3, 3],
           [4, 4],
           [5, 5],
           [6, 6],
           [7, 7],
           [8, 8]])
```

Функция F, которая выявляет количество атак ферзей, выглядит следующим образом:

```
In: def F(Q, N):
    cnt = 0
    for i in range(N):
        for j in range(i + 1, N):
            if abs(Q[i, 0] - Q[j, 0]) == abs(Q[i, 1] - Q[j, 1]):
                cnt += 1
    return cnt * 2 # учитываем взаимные атаки
```

Посмотрим, сколько у атак у исходной расстановки.

```
In: F(matrix, 8)
```

```
Out: 56
```

В нашей задачи функция G будет случайной незначительной перестановкой номеров горизонтали в исходном наборе.

```
In: def G(Q, N):
    pos = Q.copy()
    while True:
        i = np.random.randint(0, N - 1)
        j = np.random.randint(0, N - 1)
        if i != j:
            break
    pos[i, 1], pos[j, 1] = pos[j, 1], pos[i, 1]
    return pos # получаем новое расположение
```

Теперь выведем и сам метод имитации отжига.

```
In: def SA(Q, T, schedule):
    N = np.shape(Q)[0]
    x_hat = Q.copy()
```

```

while F(x_hat, N) != 0:
    x_tilda = G(x_hat, N)
    delta = F(x_tilda, N) - F(x_hat, N)
    prob = np.exp(- delta / T)
    if (delta < 0) or (prob >= np.random.random()):
        x_hat = x_tilda
    # используем метод тушения для понижения температуры
    T *= schedule
return x_hat

```

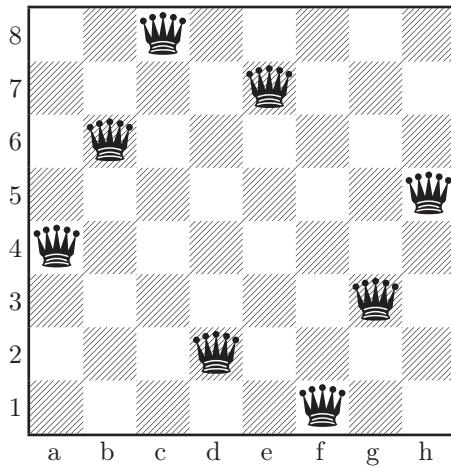


Рисунок 1.2 — Оптимальное расположение

Так для нашего примера с гиперпараметрами $T(0) = 100, \alpha = 0.9$ мы получаем следующее оптимальное решение (рис. 1.2):

In: SA(matrix, 100, 0.9)

Out: array([[1, 4],
 [2, 6],
 [3, 8],
 [4, 2],
 [5, 7],
 [6, 1],
 [7, 3],
 [8, 5]])

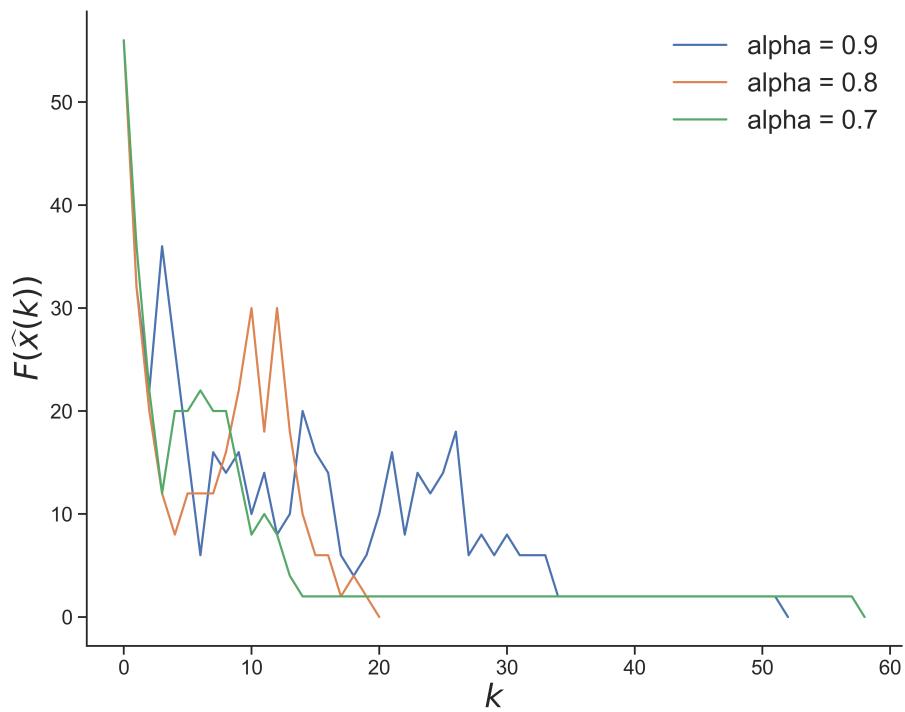


Рисунок 1.3 — Оптимизация расстановки 8 ферзей в зависимости от гиперпараметра α .

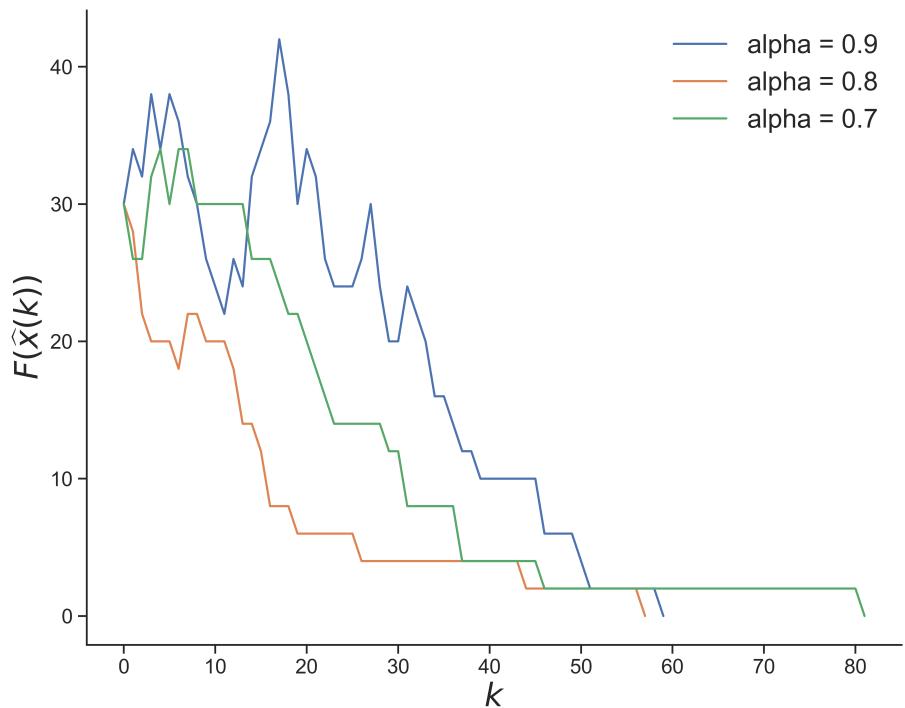


Рисунок 1.4 — Оптимизация расстановки 25 ферзей в зависимости от гиперпараметра α .

1.3. Минимизация негладкой функции

Воспользуемся алгоритмом имитации отжига для нахождения глобального минимума следующей функции:

$$f(x) = x^2(1 + |\sin 80x|).$$

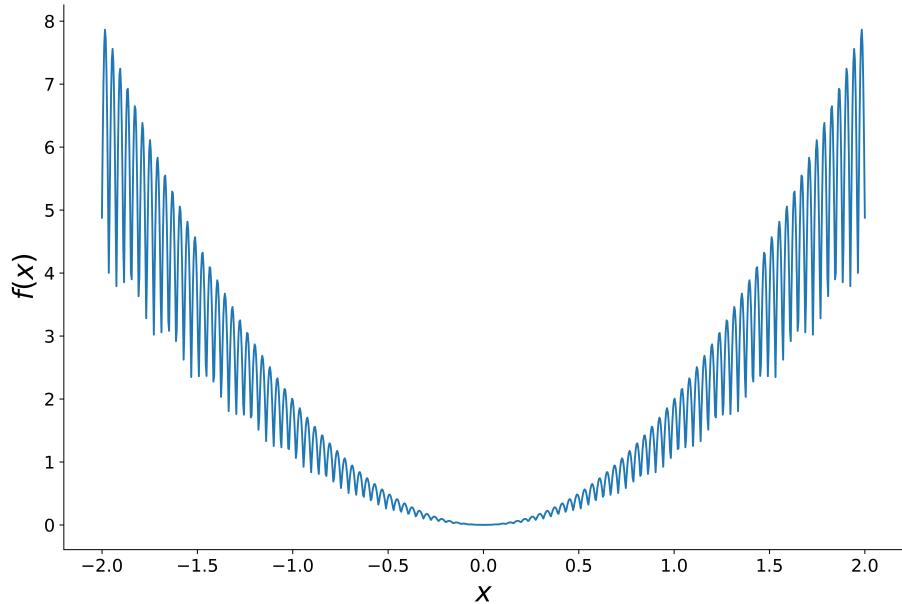


Рисунок 1.5

Стандартные методы оптимизации — к примеру, метод градиентного спуска — в данном случае не применимы. Вследствие наличия модуля эта функция не дифференцируема. Также она имеет очень большое количество локальных минимумов, что затрудняет, к примеру, мультистарт — запуск градиентного спуска из разных начальных направлений.

Применим наш алгоритм к данной задаче. Пусть $T(0) = 0.6$. Для понижения температуры будем использовать Больцмановский отжиг (1.1), а в качестве функции вероятностных распределений G — семейство нормальных распределений (1.4).

```
In: def SA(space, T, epsilon): # за space берется np.linspace(-2,2,1000)
    x_hat = np.random.choice(space)
    T_0 = T
    k = 1
    while True:
        x_tilda = np.random.normal(x_hat, T)
        delta = F(x_tilda) - F(x_hat)
        prob = np.exp(-delta / T)
```

```

if (delta < 0) or (prob >= np.random.random()):
    x_hat = x_tilda
if (x_hat < epsilon) and (x_hat > 0):
    return x_hat
T = T_0 / np.log(1 + k)
k += 1

```

Остановка итерационного процесса и скорость метода зависят от того, насколько близко мы хотим приблизиться к глобальному минимуму. Так, при точности 10^{-1} , что довольно много, для 1000 повторений алгоритма метод отжига находит глобальный минимум в среднем за 1.97e-3 секунды со стандартным отклонением в 3.47e-5 секунды. Однако, увеличив точность до 10^{-6} , среднюю скорость занимает уже 1.27 секунды со стандартным отклонением в 2.1e-5 секунды. Это наглядно представлено на рисунке 1.6.

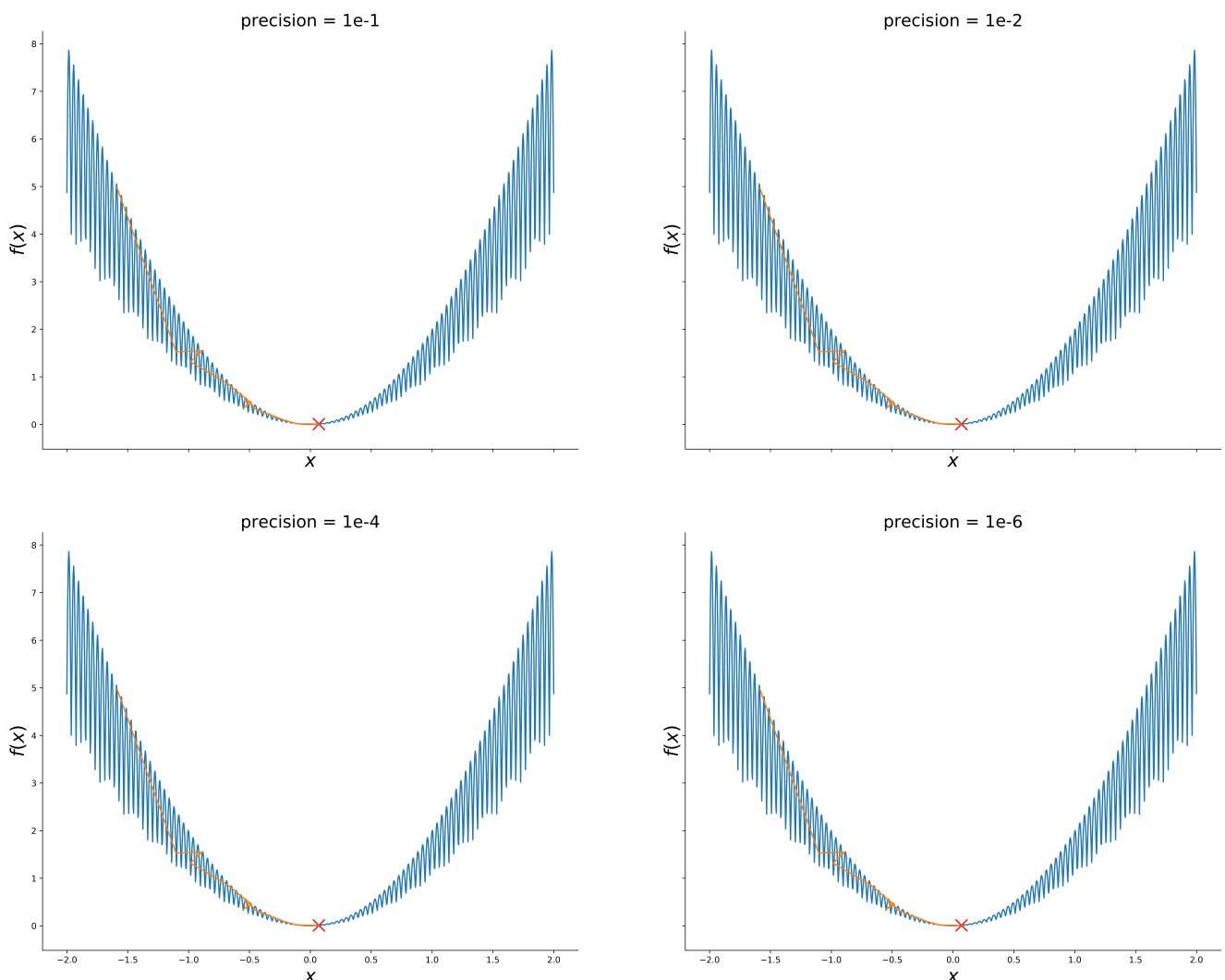


Рисунок 1.6 — Оптимизационный процесс в зависимости от точности.

1.4. Задача коммивояжера

Задача коммивояжера (*Traveling Salesman Problem*, TSP) является образцовым методом проверки многих оптимизационных алгоритмов и заключается в поиске кратчайшего маршрута между городами. Путь должен быть проложен так, чтобы маршрут единственно проходил через все города и его конечная точка совпадала с изначальной.

TSP имеет множество приложений в планировании и логистике, а также выступает в качестве подзадачи во многих других областях. В таком случае города могут представлять, к примеру, клиентов, а расстояние между городами — время или стоимость путешествия.

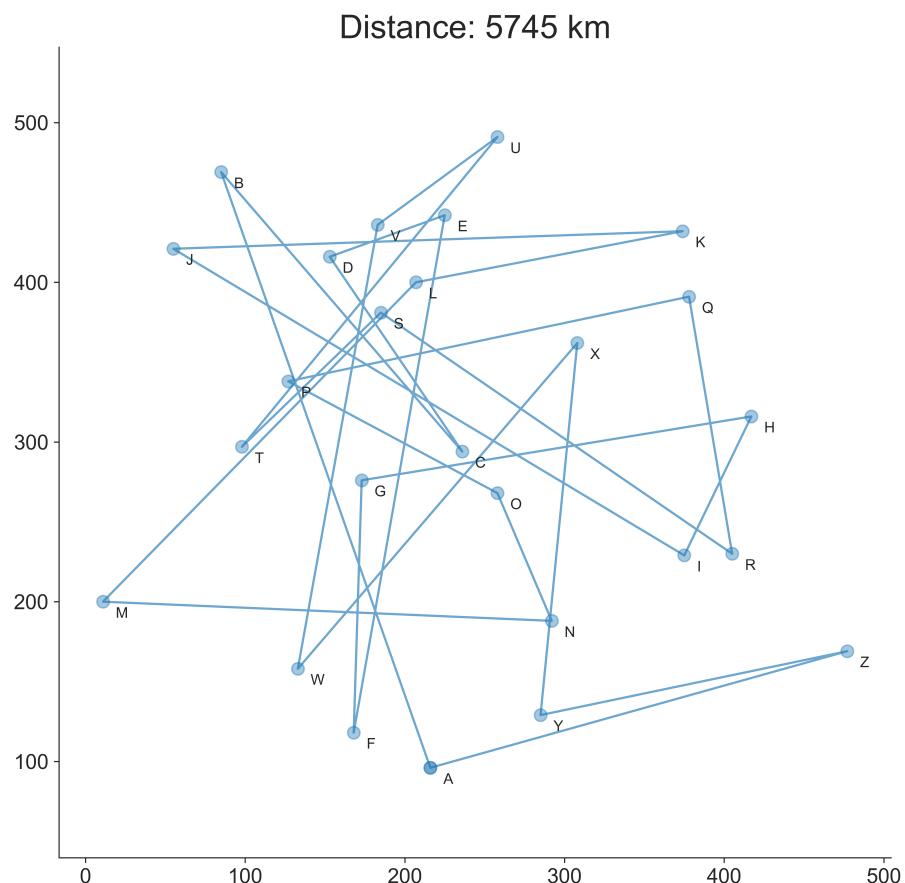


Рисунок 1.7 — Изначальный маршрут для 26-ти городов.

Для нашего примера создадим карту. Функция `map_city` будет принимать желаемое количество городов в качестве входных данных и выдавать два списка, из которых далее создается общий список кортежей. Первым элементом кортежа является наименование города, а вторым — его расположение в декартовой системе координат.

```
In: def map_city(cities_num):
    letters = [chr(i) for i in range(65, 65 + cities_num)]
    coord = np.random.randint(1, 500, size=(cities_num, 2))
    return letters, coord
```

Наш маршрут будет состоять из 26-ти городов.

```
In: names, cities = map_city(26)
store_val = list(zip(names, cities))
```

Определим расстояние от города i до всех остальных посредством функции `distance_dict`. Для измерения расстояния между городами будем использовать евклидову метрику. Напомним, что евклидово расстояние между точками $x = (x_1, \dots, x_d)$ и $u = (u_1, \dots, u_d)$ задается как:

$$\rho(x, u) = \sqrt{\sum_{i=1}^d (x_i - u_i)^2}$$

```
In: def distance_dict(cities, n):
    d = dict()
    for i in range(n):
        city = dict()
        for j in range(n):
            if i == j:
                continue
            c_a = cities[i][1]
            c_b = cities[j][1]
            dist = np.sqrt((c_a[0] - c_b[0])**2 + (c_a[1] - c_b[1])**2)
            city[cities[j][0]] = dist
        d[cities[i][0]] = city
    return d
```

```
In: cities_d = distance_dict(store_val, len(store_val))
```

Функция `F` для подсчета общего расстояния путешествия:

```
In: def F(path, cities):
    dist = 0
    for i in range(len(path) - 1):
        dist += cities[path[i]][path[i + 1]]
    dist += cities[path[-1]][path[0]]
    return dist
```

За функцию `G` будет выступать простая перестановка как и в задаче о N ферзях.

```
In:  def G(path, n):
      pos = path.copy()
      while True:
          i = np.random.randint(0, n - 1)
          j = np.random.randint(0, n - 1)
          if i != j:
              break
          pos[i], pos[j] = pos[j], pos[i]
      return pos
```

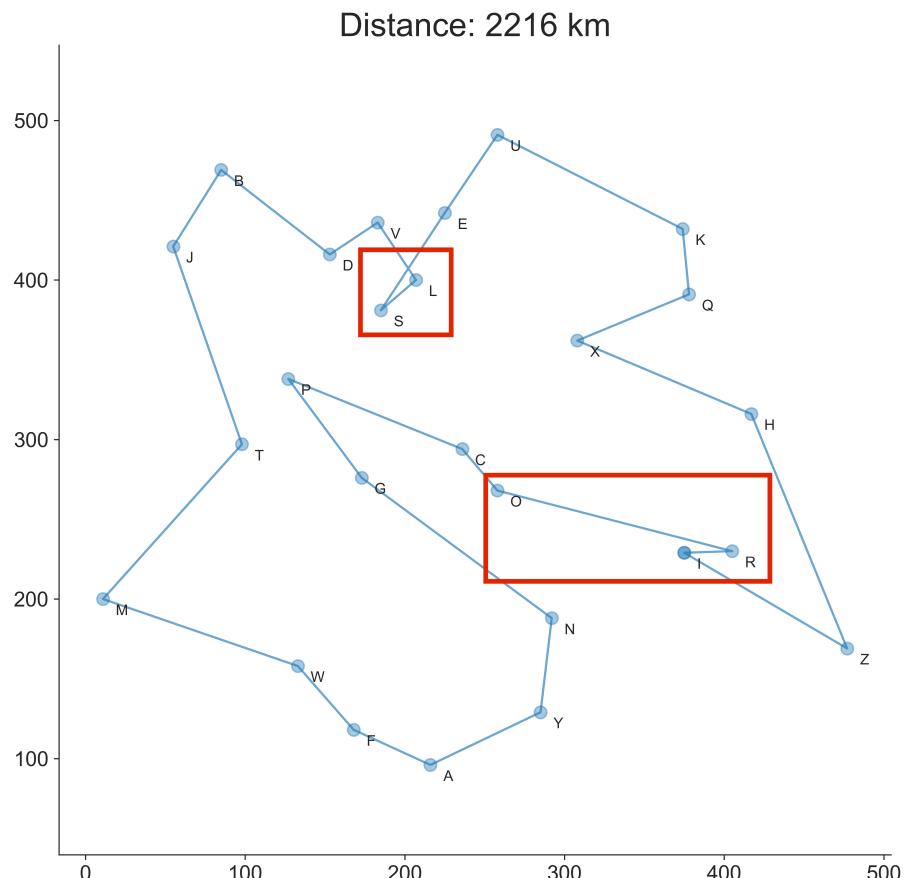


Рисунок 1.8 — Применение метода отжига для построения оптимального маршрута для 26-ти городов.

Для понижения температуры используем Больцмановский отжиг (1.1).

```
In:  def SA(path, T):
      path_hat = path
      n = len(path_hat)
      np.random.shuffle(path_hat)
      T_0 = T
      k = 1
      for i in range(100000):
          path_tilda = G(path_hat, n)
          delta = F(path_tilda, cities_d) - F(path_hat, cities_d)
```

```
prob = np.exp(- delta / T)
```

Теперь построим оптимальный маршрут (рис. 1.8).

```
In: path_opt = SA(names, 100)
```

Несмотря на то, что метод отжига сократил преодолеваемую дистанцию, TSP была решена неидеально. К примеру, можно выделить соединение вершин E-S-L-V: очевидно, оно не оптимально, поскольку путь E-L-S-V имеет меньшее расстояние. Тем не менее, само расположение городов весьма удовлетворительно.

1.5. Вывод

Рассмотрев метод отжига, выявим его плюсы и минусы.

Таблица 1.1 — Метод отжига

Преимущества	Недостатки
1. Имеет простую реализацию	1. Не подходит для задач с небольшим количеством локальных минимумов
2. Не застревает в локальных минимумах	2. Не всегда сходится к решению
3. Для сложных задач (наподобие TSP) дает вполне приемлемое решение	

2. Метод роения частиц

Метод роения частиц (*Particle Swarm Optimization*, PSO) является одним из алгоритмов коллективной оптимизации и основывается на имитации социального поведения в колонии живых организмов — к примеру, стаи птиц или колонии муравьев, — выполняющих коллективный поиск места с наилучшими условиями для существования. При поиске пищи каждая особь колонии передвигается по окружающей среде независимо от остальных организмов с некой долей случайности в своих движениях. Рано или поздно одна из особей находит пропитание и, будучи социальным организмом, сообщает об этом остальным, что стягивает ее соседей к данной пище.

2.1. Алгоритм

Найдем глобальный экстремум функции $F: \mathbb{R}^n \rightarrow \mathbb{R}$. Для определенности будем искать глобальный минимум:

$$F(x_i) \rightarrow \min_{x_i \in \mathbb{R}^n}$$

Пусть в нашем рое существует ℓ частиц, тогда рой имеет вид $x = \{x_i\}_{i=1}^\ell$, $x_i \in U \subseteq \mathbb{R}^n$. Пусть также определен вектор скорости $v = \{v_i\}_{i=1}^\ell$, i -я компонента которого является скоростью i -й частицы, $v_i \in U \subseteq \mathbb{R}^n$.

1. Изначально случайным образом выбираем расположение роя $x(0)$ и скорость движения каждой частицы $v(0)$.
2. Определяем новое расположение роя:

$$x(t+1) \equiv x(t) + v(t)$$

3. Выбираем наилучшую точку для i -й частицы:

$$p_i(t) = \begin{cases} x_i(t), & F(x_i(t+1)) \geq F(x_i(t)), \\ x_i(t+1), & F(x_i(t+1)) < F(x_i(t)) \end{cases}, \quad 1, \dots, \ell \quad (2.1)$$

Тогда вектор наилучших позиций для каждой частицы имеет вид:

$$p(t) = \{p_i(t)\}_{i=1}^\ell$$

4. Выбираем наилучшую точку для всего сообщества:

$$g(t) \equiv \underset{i \in (1, \dots, \ell)}{\operatorname{argmin}} F(p_i(t)) \quad (2.2)$$

5. Обновляем скорость для i -й частицы посредством следующей формулы:

$$v_i(t+1) \equiv wv_i(t) + c_1\xi_1(t)[p_i(t) - x_i(t)] + c_2\xi_2(t)[g(t) - x_i(t)], \quad i = 1, \dots, \ell \quad (2.3)$$

где $w \in \mathbb{R}$ — инерционный вес, $c_1, c_2 \in \mathbb{R}$ — коэффициенты ускорения, $\xi_1, \xi_2 \sim U(0, 1)$.

Тогда вектор скорости имеет вид:

$$v(t+1) = \{v_i(t+1)\}_{i=1}^{\ell}$$

Три фактора влияют на частицу в положении $x_i(t)$. С одной стороны, когнитивное воздействие побуждает частицу двигаться к ее лучшей позиции $p_i(t)$, с другой стороны — социальное воздействие побуждает частицу продвигаться в сторону лучшей позиции роя $g(t)$. Кроме того, собственная скорость $v_i(t)$ обеспечивает движение по инерции, что позволяет частице преодолевать локальные минимумы и исследовать неизвестные области заданного пространства. Таким образом, происходит переход от точки $x_i(t)$ в точку $x_i(t+1)$, что представлено на следующем графике:

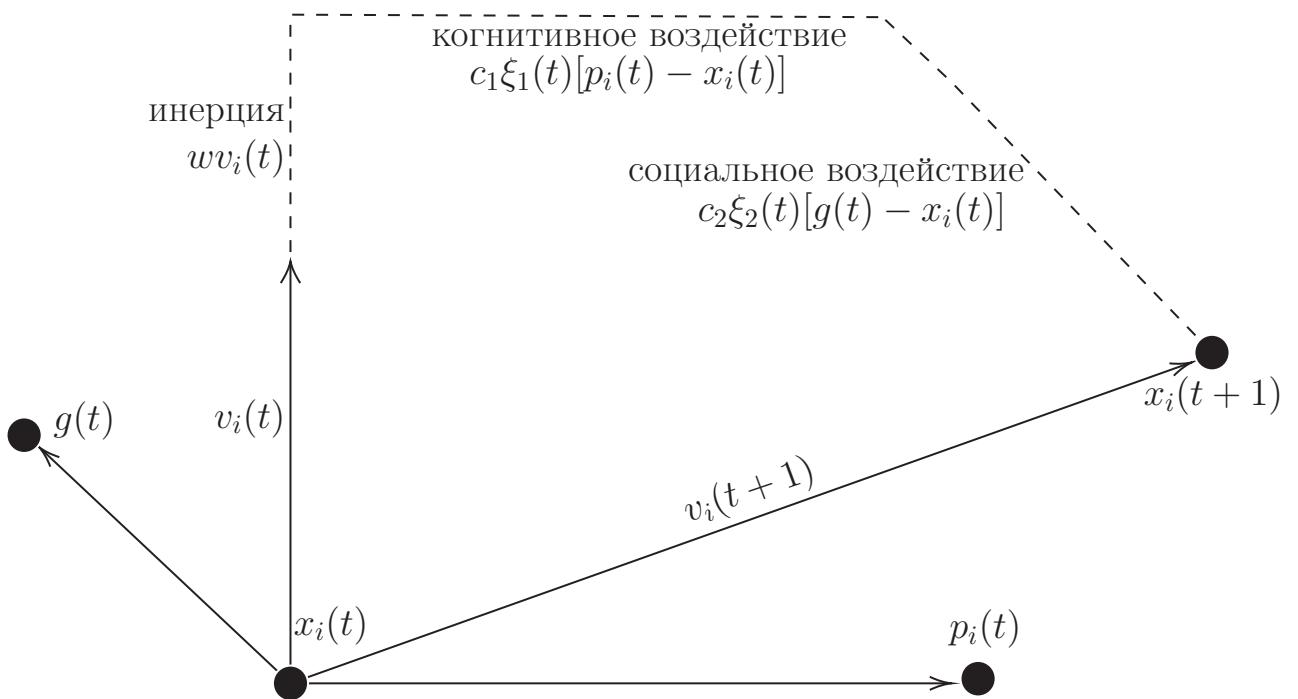


Рисунок 2.1

Алгоритм использует две последовательности равномерно распределенных случайных величин $\xi_1(0), \dots, \xi_1(t)$ и $\xi_2(0), \dots, \xi_2(t)$, которые масштабируются

по константам c_1 , c_2 . Данные константы влияют на максимальный размер шага, который частица может сделать за одну итерацию. При $c_1 = 0$ метод роения частиц будет опираться только на наилучшую позицию сообщества — в таком случае алгоритм будет быстро сходиться, однако маловероятен факт нахождения глобального оптимума. При $c_1 > 0$ метод использует связь всего сообщества — скорость конвергенции падает, но глобальный оптимум оказывается более вероятным.

6. Возвращаемся к 2 шагу, пока не достигнем оптимума.

2.2. Функция Розенброка

Функция Розенброка — это невыпуклая функция вида

$$f(x_1, \dots, x_n) = \sum_{i=1}^{n-1} [(a - x_i)^2 + b(x_{i+1} - x_i^2)^2]$$

использующаяся в качестве оценки производительности оптимизационных алгоритмов. Для наших целей будем использовать трехмерный случай:

$$f(x, y) = (a - x)^2 + b(y - x^2)^2,$$

где глобальный достигается в точке (a, a^2) , где $f(a, a^2) = 0$.

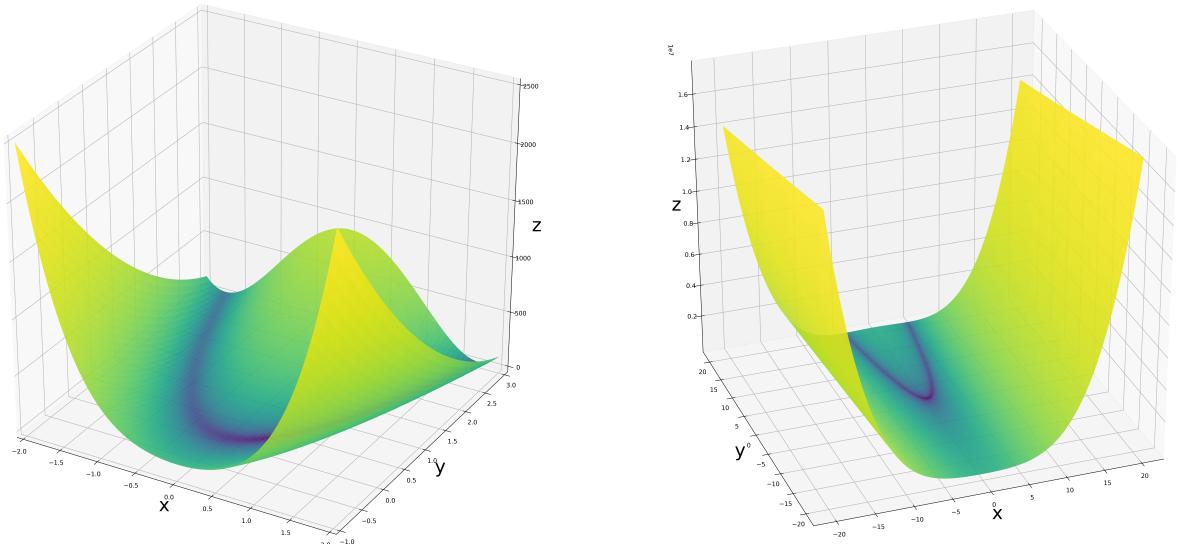


Рисунок 2.2 — Функция Розенброка с параметрами $a = 1$, $b = 100$.

Обычно $a = 1$, $b = 100$. Тогда функция Розенброка примет следующий вид (рис. 2.2):

$$f(x, y) = (1 - x)^2 + 100(y - x^2)^2$$

Глобальный минимум данной функции находится внутри «долины»: в нашем случае — в точке $(1, 1)$. Найти долину достаточно легко, однако приблизиться к глобальному минимуму, считается, довольно сложно.

Только при $a = 0$, функция является симметричной, а ее минимум находится в начале координат.

Определим функцию Розенброка:

```
In: def func(x):  
    return (1 - x[0]) ** 2 + 100 * (x[1] - x[0] ** 2) ** 2
```

Инициализируем метод роения частиц, используя объектно-ориентированное программирование. В первую очередь создадим класс частиц.

```

In:  class Particle:
        def __init__(self, arg, space):
            self.pos = np.asarray([])          # расположение частицы
            self.velocity = np.asarray([])     # вектор скорости частицы
            self.pos_best = None              # лучшее расположение
            for i in range(arg):
                pos_i = np.random.uniform(space[i][0], space[i][1])
                self.pos = np.append(self.pos, pos_i)
                vel_i = np.random.uniform(0.2*space[i][0], 0.2*space[i][1])
                self.velocity = np.append(self.velocity, vel_i)
            # pos_best --- это список, состоящий из лучшего расположения
            # частицы и значения функции в данной точке
            self.pos_best = [self.pos.copy(), func(self.pos)]

        def update_position(self):
            self.pos += self.velocity

        def update_velocity(self, w, c1, c2, swarm_best):
            inertion = w * self.velocity
            xi_1 = np.random.uniform()
            xi_2 = np.random.uniform()
            cognitive_acceler = c1 * xi_1 * (self.pos_best[0] - self.pos)
            social_acceler = c2 * xi_2 * (swarm_best - self.pos)
            self.velocity = inertion + cognitive_acceler + social_acceler

        def choose_personal_best(self):
            if func(self.pos) < func(self.pos_best[0]):
                self.pos_best[0] = self.pos.copy()
                self.pos_best[1] = func(self.pos)

```

Теперь создадим класс ParticleSwarmOptimisation. Поскольку инерционный вес w должен быть близким к 1, а коэффициенты ускорения c_1, c_2 должны быть достаточно малыми, при инициализации класса установим гиперпараметры по умолчанию равными следующим величинам: $w = 1.0, c1 = 0.2, c2 = 0.2$. Посредством метода search_global будем искать глобальный минимум функции Розенброка.

```

In:  class ParticleSwarmOptimisation:
      def __init__(self, ell=40, w=1.0, c1=0.2, c2=0.2, max_iter=1000,
                  tol=1e-6):
          """
          PARAMETERS:
          ell --- количество частиц в рое.
          w --- инерционный вес.
          c1 --- коэффициент ускорения когнитивного воздействия на частицу.
          c2 --- коэффициент ускорения социального воздействия на частицу.
          max_iter --- максимальное количество итераций.
          tol --- точность.
          """
          self.ell = ell
          self.w = w
          self.c1 = c1
          self.c2 = c2
          self.max_iter = max_iter
          self.tol = tol
          self.swarm_best = None # лучшее расположение для всего роя
          self.swarm = None       # расположение всех частиц (рой)

      def search_global(self, arg, space):
          """
          PARAMETERS:
          arg --- количество аргументов функции.
          space --- область поиска оптимума. Задается как список из кортежей, где кортеж --- это область значений одного аргумента функции.
          """
          self.arg = arg
          self.space = np.array(space)
          self.swarm = np.asarray([])

          # генерируем расположение роя
          for _ in range(self.ell):
              self.swarm = np.append(self.swarm,
                                    Particle(self.arg, self.space))

          for k in range(self.max_iter):

```

```

        for i in range(self.ell):
            # обновляем расположение частицы
            self.swarm[i].update_position()
            # сравниваем с лучшей точкой частицы
            self.swarm[i].choose_personal_best()

        # выбираем лучшую точку для роя
        if k != 0:
            dist_0 = self.dist(self.swarm_best[0])
            self.choose_social_best()
            dist_1 = self.dist(self.swarm_best[0])

        # останавливаем поиск в условиях заданной точности
        if (dist_0 != dist_1) and (abs(dist_0-dist_1) <= self.tol):
            break
        else:
            self.choose_social_best()

        # обновляем вектор скорости
        for i in range(self.ell):
            self.swarm[i].update_velocity(self.w, self.c1,
                                         self.c2, self.swarm_best[0])

    print(f"Глобальный оптимум: {self.swarm_best[0]}")
    print(f"Значение функции в данной точке: {self.swarm_best[1]}")

    def choose_social_best(self):
        best = min([[self.swarm[i].pos_best[0],
                     self.swarm[i].pos_best[1]] for i in range(self.ell)],
                   key=lambda x: x[1])
        self.swarm_best = best

    def dist(self, x):
        return np.sqrt(np.sum(x ** 2))

```

Инициализируем рой с 100 частицами.

```
In: sw = ParticleSwarmOptimisation(w=0.95, ell=100, tol=1e-20)
```

Найдем глобальный минимум. Перемещение роя в поисках глобального оптимума представлено на рисунке 2.3.

```
In: np.random.seed(53)
sw.search_global(2, [(-10, 10), (-10, 10)])
```

Out: Глобальный оптимум: [1. 1.] .

Значение функции в данной точке: 6.366439023928984e-22.

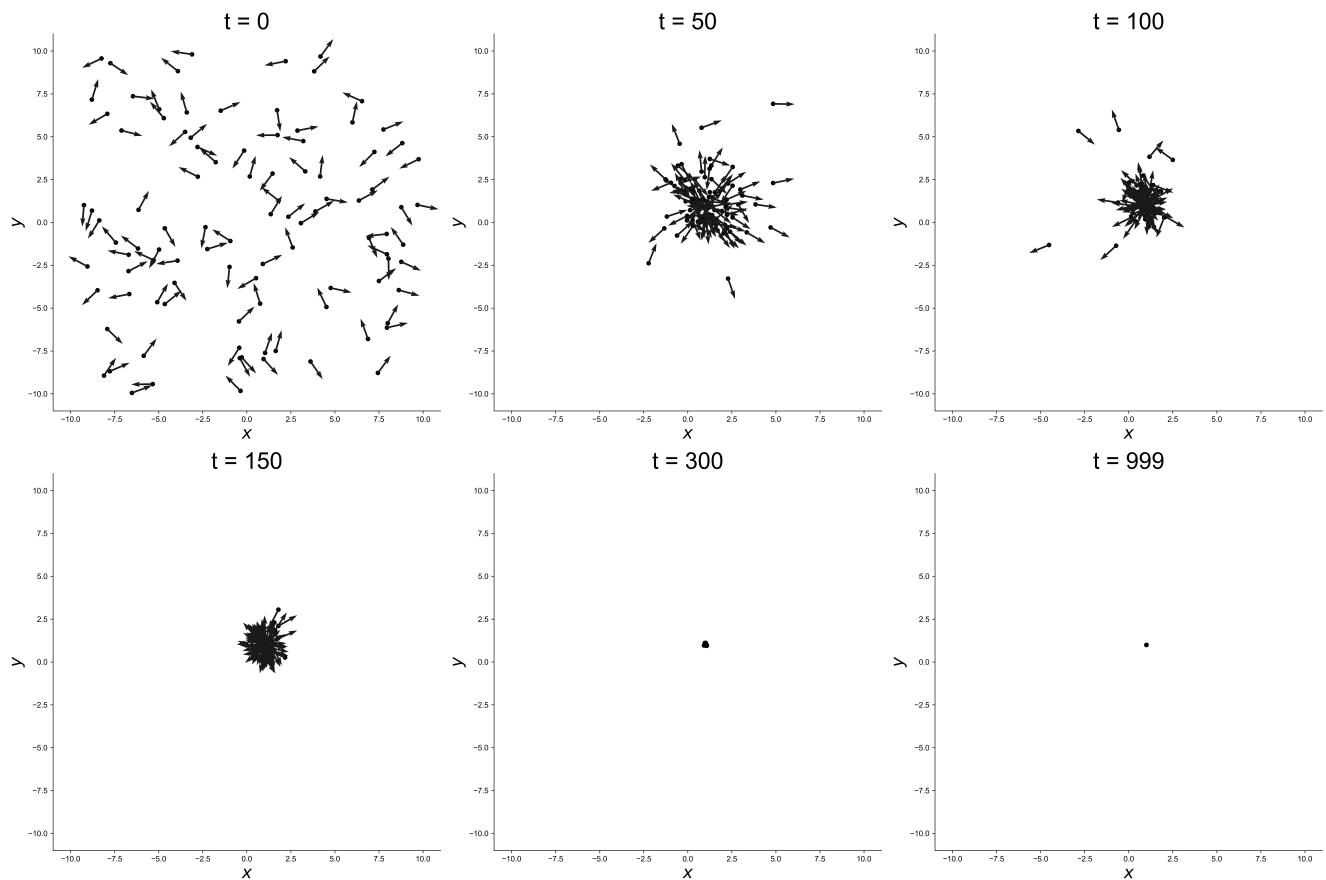


Рисунок 2.3 — Расположение роя в разные моменты времени.

Из графиков на рисунке 2.4 видно, что за 1000 итераций метод роения частиц сходится к глобальному минимуму функции Розенборока с экспоненциальной скоростью с точностью 6.37e-22. Средняя скорость реализации алгоритма составляет 1.79 секунды со стандартным отклонением в 0.111 секунд.

2.3. Вывод

Рассмотрев метод роения частиц, выявим его плюсы и минусы.

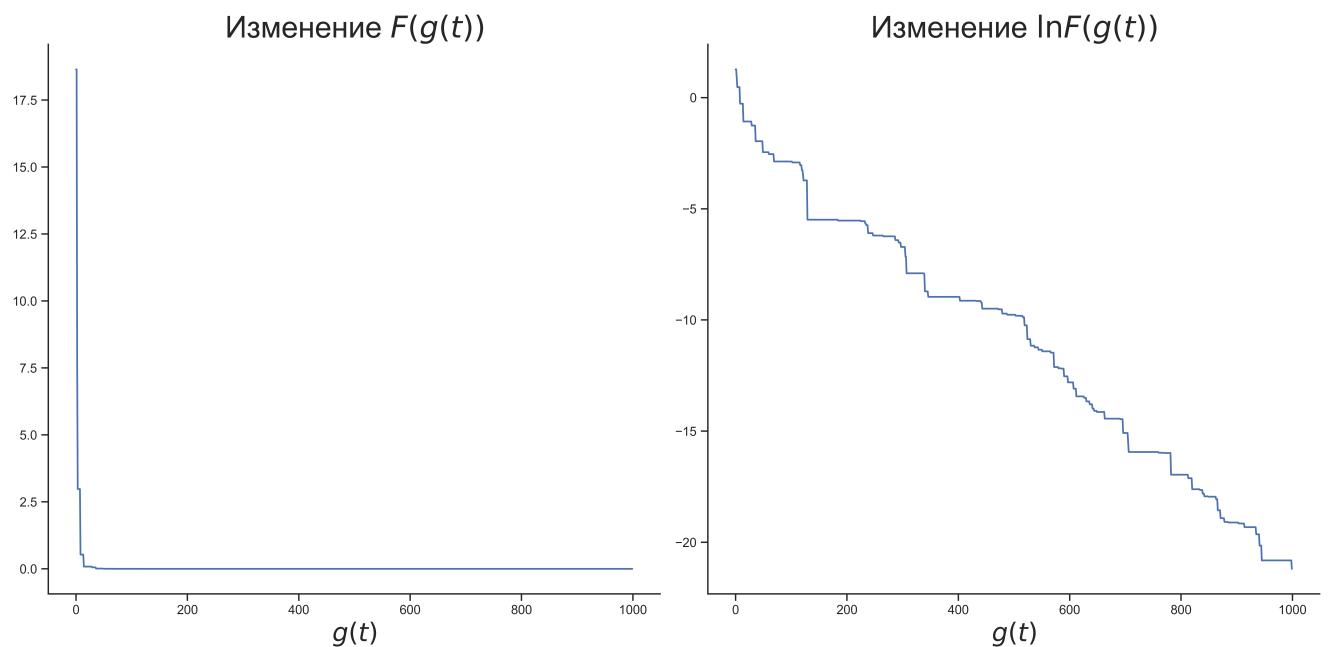


Рисунок 2.4 — Оптимизационный процесс.

Таблица 2.1 — Метод роения частиц

1.

2.

3.

1.

2.

3. Генетический алгоритм

Генетический алгоритм (*Genetic Algorithm*, GA) является одним из самых популярных типов эволюционных алгоритмов, которые — как можно понять из их названия, — используют теорию эволюции, в частности теорию Дарвина, в решении той или иной оптимизационной задачи. Суть алгоритма заключается в поиске наилучшего решения по принципу естественного отбора: случайным образом создается поколение, в ходе эволюции которого происходит скрещивание генов — таким образом, появляется новое поколение (старые особи создают потомков) — и их мутация.

3.1. Алгоритм

Традиционно в качестве решения генетический алгоритм использует бинарное кодирование (binary encoding). Однако в данной работе будет проиллюстрирован метод реального кодирования (real value encoding), то есть решение будет представляться в вещественных числах.

Рассмотрим задачу нахождения глобального минимума функции $F: \mathbb{R}^n \rightarrow \mathbb{R}$:

$$\operatorname{argmin}_{x \in \mathbb{R}^n} f(x),$$

где каждая особь кодируется вектором $x = (x_1, \dots, x_n)$, называемым хромосомой, компоненты которой являются генами.

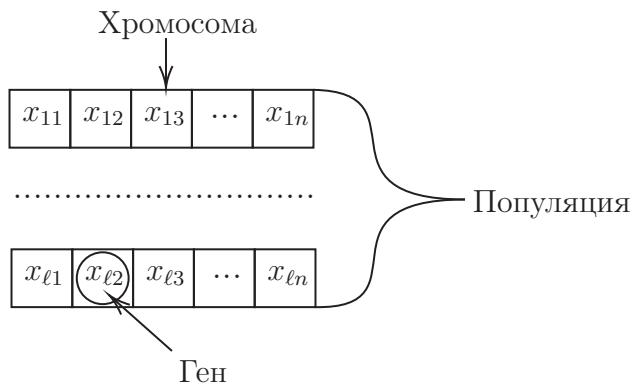


Рисунок 3.1

Пусть у нашего поколения, или популяции, P есть ℓ особей. Тогда поколение в момент времени t примет следующий вид:

$$P_t = \{x_{i,t}\}_{i=1}^\ell,$$

где у каждой хромосомы имеется n генов: $x_{i,t} = (x_{i1,t}, \dots, x_{in,t})$

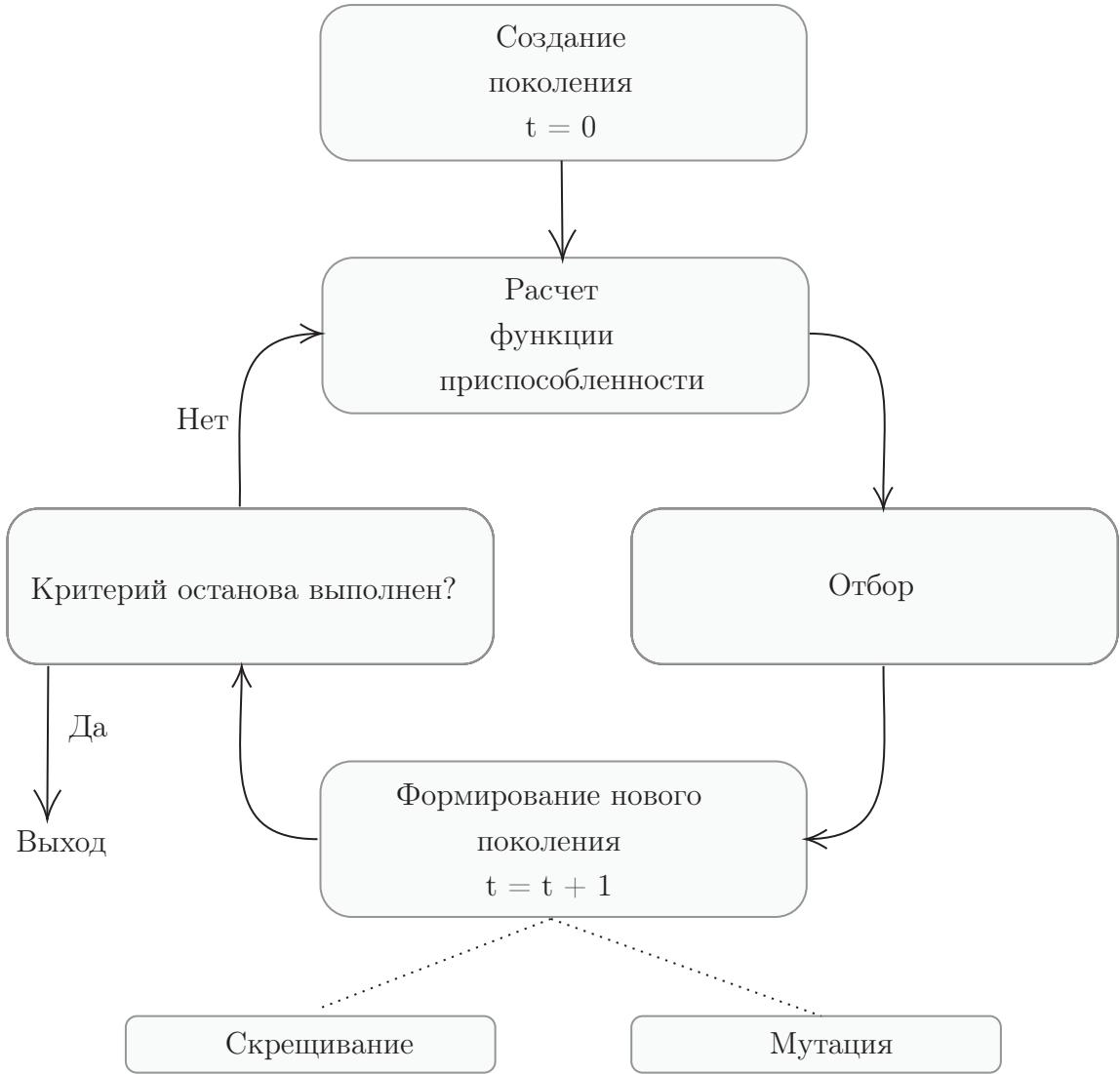


Рисунок 3.2 — Генетический алгоритм.

Теперь рассмотрим сам алгоритм, который также представлен на рисунке 3.2:

1. Генерируем первое поколение особей случайным образом:

$$P_0 = \{x_{1,0}, \dots, x_{\ell,0} \mid (x_{i1,0}, \dots, x_{in,0}) \in D, x_{ij,0} \sim U(a, b)\},$$

где $D = \{(x_1, \dots, x_n) \in \mathbb{R}^n \mid a < x_i < b, a, b \in \mathbb{R}\}$ — заданная область оптимизации (гиперкуб или гиперпрямоугольник).

2. Оцениваем приспособленность (fitness) текущего поколения.

Сначала вычисляем значения целевой функции для каждой особи:

$$F_t = \{f(x_{i,t})\}_{i=1}^{\ell}$$

Затем используем такое преобразование σ , что все компоненты F_t будут лежать на отрезке $[0, 1]$:

$$\sigma(z) = \frac{z - F^{worst}}{F^{best} - F^{worst}}, \quad (3.1)$$

где F^{best} , F^{worst} — лучшее и худшее значения целевой функции в текущем поколении соответственно.

Таким образом, получим значения приспособленности особей:

$$F'_t = \{\sigma(f(x_{i,t}))\}_{i=1}^{\ell}$$

3. Отберем k наиболее приспособленных особей для дальнейшего размножения.

Будем считать, что особь x' более приспособлена, чем особь x'' , если

$$\sigma(f(x')) > \sigma(f(x''))$$

Ранжируем особи в текущей популяции по значениям их приспособленности в порядке убывания, то есть от самых приспособленных к самым неприспособленным:

$$\sigma(f(x_t^{(1)})) \geq \dots \geq \sigma(f(x_t^{(k)})) \geq \dots \geq \sigma(f(x_t^{(\ell)}))$$

4. Скрещиваем (crossover) k наиболее приспособленных особей со всеми остальными. Так мы распространяем «хорошие» гены по популяции.

Под скрещиванием будем иметь в виду операцию создания новой особи, — более точно ее хромосомы, — у которой часть генов будет от особи x с вероятностью $\mathbb{P}(z_i = x_i)$, а часть от особи y с вероятностью $\mathbb{P}(z_i = y_i)$.

$$\exists x_t, y_t \in D: \Psi(x_t, y_t) = z_{t+1} = (z_{1,t+1}, \dots, z_{n,t+1}), \quad (3.2)$$

где Ψ — оператор скрещивания особей x и y для получения особи z .

$$z_i = \begin{cases} x_i, & \text{с вероятностью } \mathbb{P}(z_i = x_i) = \frac{\sigma(f(x))}{\sigma(f(x)) + \sigma(f(y))} \\ y_i, & \text{с вероятностью } \mathbb{P}(z_i = y_i) = \frac{\sigma(f(y))}{\sigma(f(x)) + \sigma(f(y))} \end{cases}$$

Также заметим, что, во-первых, $\mathbb{P}(z_i = x_i) + \mathbb{P}(z_i = y_i) = 1$, а во-вторых, гены более приспособленной особи чаще присваиваются, нежели случайно выбранной особи. Это объясняется тем, что вероятность получения гена приспособленной особи превышает вероятность получения гена случайно выбранной особи в большинстве случаев.

5. Некоторых особей в текущей популяции — кроме лучшей, то есть $x_t^{(1)}$, — подвергнем мутации. Оператор мутации меняет произвольное количество генов в хромосоме особи на другие — однако довольно близкие к исходным, — гены.

$$\Upsilon(x_t) = \hat{x}_{t+1}, \quad (3.3)$$

где Υ — оператор мутации, x — мутируемая особь.

Если ген в хромосоме подвергается мутации, то он принимает следующий вид:

$$\hat{x}_i = x_i + \delta_i \in D$$

6. Переходим к пункту 2, пока не выполнен критерий останова.

Приведем несколько критериев останова:

- а) Превышение заданного количества поколений (количества эволюций популяции).
- б) Достигнута необходимая точность приближения.

3.2. Функция Экли

Функция Экли также, как и функция Розенброка, используется в качестве оценки производительности оптимизационных алгоритмов. Она имеет следующий вид:

$$f(x_1, \dots, x_n) = 20 + e - 20 \exp \left\{ -0.2 \sqrt{\frac{1}{n} \sum_{i=1}^n x_i^2} \right\} - \exp \left\{ \frac{1}{n} \sum_{i=1}^n \cos(2\pi x_i) \right\}$$

Будем рассматривать трехмерный случай. Тогда глобальный минимум функции Экли достигается в точке $(x, y) = (0, 0)$ со значением целевой функции $f(0, 0) = 0$ (рис. 3.3).

Воспользуемся принципами объектно-ориентированного программирования для решения данной задачи.

Определим функцию Экли в Python и найдем ее глобальный минимум.

```
In: def ackley_func(x):
    part_1 = 20*np.exp(-0.2*np.sqrt(0.5*(x[0]**2 + x[1]**2)))
    part_2 = np.exp(0.5*(np.cos(2*np.pi*x[0]) + np.cos(2*np.pi*x[1])))
    return 20 + np.exp(1) - part_1 - part_2
```

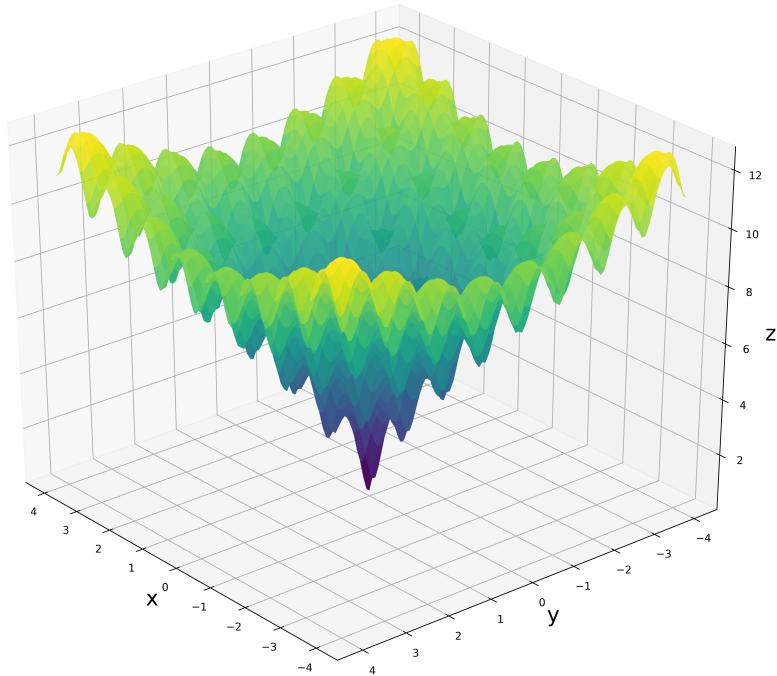


Рисунок 3.3 — Функция Экли в трехмерном пространстве.

Сначала создадим класс Individual, который моделирует каждую особь в нашей популяции: присваивает каждой особи хромосому, ее длину, гены, а также значение целевой функции в точке, использующей хромосому данной особи, значение приспособленности особи. Также определим названия для каждой особи в популяции для облегчения восприятия данного класса.

```
In:  class Individual:
    def __init__(self, search_space, chromosome_len):
        self.search_space = search_space
        self.chromosome_len = chromosome_len
        self.chromosome = np.array([self.create_gene(j)
                                    for j in range(chromosome_len)])
        self.target_value = None # значение целевой функции при
        # использовании данной особи
        self.fitness = None # значение приспособленности особи
        # относительно других

        self.name = '#' + ''.join(map(str,
                                       np.random.randint(0, 9, 7).tolist()))

    def create_gene(self, pos):
        return np.random.uniform(self.search_space[pos][0],
                               self.search_space[pos][1])
```

```

def __repr__(self):
    chr = '; '.join(list(map(str,
                             self.fittest_individual.chromosome.tolist())))
    tar = self.target_value
    return f'{self.name}: chromosome = {(chr)}; target_value = {tar}'

```

Далее определим генетический алгоритм под классом GeneticAlgorithm. Гиперпараметрами нашего метода являются: количество особей в популяции (ell), количество отбираемых особей для скрещивания (k), максимальная мутация гена (mutation_rate) и критерий останова в качестве превышения заданного количества поколений (max_iter).

```

In: class GeneticAlgorithm:
    def __init__(self, ell=1000, k=200, mutation_rate=0.1,
                 max_iter=100):
        """
        PARAMETERS:
        ell --- количество особей в поколении
        k --- количество особей для размножения
        mutation_rate --- максимальная мутация гена
        max_iter --- максимальное количество новых поколений
        """
        self.ell = ell
        self.k = k
        self.mutation_rate = mutation_rate
        self.max_iter = max_iter

        self.search_space = None
        self.chromosome_len = None # длина хромосомы особи
        self.best_individuals = None # k наиболее
        # приспособленных особей
        self.fittest_individual = None # самая приспособленная особь
        self.population = None

    def search_global(self, search_space, func):
        """
        INPUT:
        search_space --- область поиска оптимума. Задается как список
        из кортежей, где кортеж --- это область значений одного
        аргумента функции
        """
        self.search_space = np.array(search_space)
        self.chromosome_len = len(self.search_space)
        self.best_target_value_history = []

```

```

# создаем первое поколение
self.population = self.create_population(self.search_space)

# проходим этапы эволюции
for i in tqdm(range(self.max_iter)):
    # оцениваем приспособленность наших особей
    self.evaluate_population(func)
    # отбираем к наиболее наиболее приспособленных особей
    self.selection()

    # формируем новые поколения
    # скрещиваем особи
    for idx in range(self.k, self.ell):
        # случайно выбираем одну из лучших особей
        agent = np.random.choice(self.best_individuals)
        select_fitted_individual = agent
        # создаем потомка и заменяем им старую особь
        offspring = self.crossover(select_fitted_individual,
                                    self.population[idx])
        self.population[idx].chromosome = offspring

    # мутируем все особи, кроме самой приспособленной
    for individual in self.population[1:]:
        self.mutate(individual)

    return self.fittest_individual

def create_population(self, search_space):
    """
    INPUT:
    search_space --- область поиска оптимума. Задается как список
    из кортежей, где кортеж --- это область значений одного
    аргумента функции
    """
    self.search_space = np.array(search_space)
    self.chromosome_len = len(self.search_space)

    return np.array([Individual(self.search_space, self.chromosome_len)
                   for i in range(self.ell)])

def evaluate_population(self, func):
    """
    INPUT:
    func --- оптимизируемая функция, которая принимает список
    в качестве аргументов
    """

```

```

F = []

for individual in self.population:
    individual.target_value = func(individual.chromosome)
    F.append(individual.target_value)

for individual in self.population:
    individual.fitness = self.normalize(individual.target_value,
                                         min(F), max(F))

def normalize(self, z, F_best, F_worst):
    """
    Нормализует значения целевой функции
    """

    INPUT:
    z --- масштабируемое значение
    F_best --- лучшее значение целевой функции
    F_worst --- худшее значение целевой функции
    """

    return (z - F_worst) / (F_best - F_worst)

def selection(self):
    """
    Оператор отбора
    """

    self.population = sorted(self.population,
                            key=lambda individual: individual.fitness,
                            reverse=True)
    self.best_individuals = self.population[:self.k]
    self.fittest_individual = self.population[0]

def crossover(self, parent_fitted, parent_random):
    """
    Оператор скрещивания

    INPUT:
    parent_fitted --- одна из самых приспособленных особей
    parent_random --- случайная особь
    """

    return np.array([parent_random.chromosome[j]
                    if np.random.uniform(0, 1) < parent_random.fitness
                    else parent_fitted.chromosome[j]
                    for j in range(parent_fitted.chromosome_len)])

```

```

def mutate(self, individual):
    """
    Оператор мутации

    INPUT:
    individual --- особь, подвергающаяся мутации
    """

    individual_hat_chromosome = np.asarray([])

    for j in range(individual.chromosome_len()):
        delta = np.random.uniform(-self.mutation_rate,
                                   self.mutation_rate)
        j_hat = individual.chromosome[j] + delta
        # на случай, если ген выйдет за пределы гиперпрямоугольника
        j_hat = min(max(j_hat, self.search_space[j][0]),
                    self.search_space[j][1])
        individual_hat_chromosome = np.append(individual_hat_chromosome,
                                                j_hat)

    individual.chromosome = individual_hat_chromosome

```

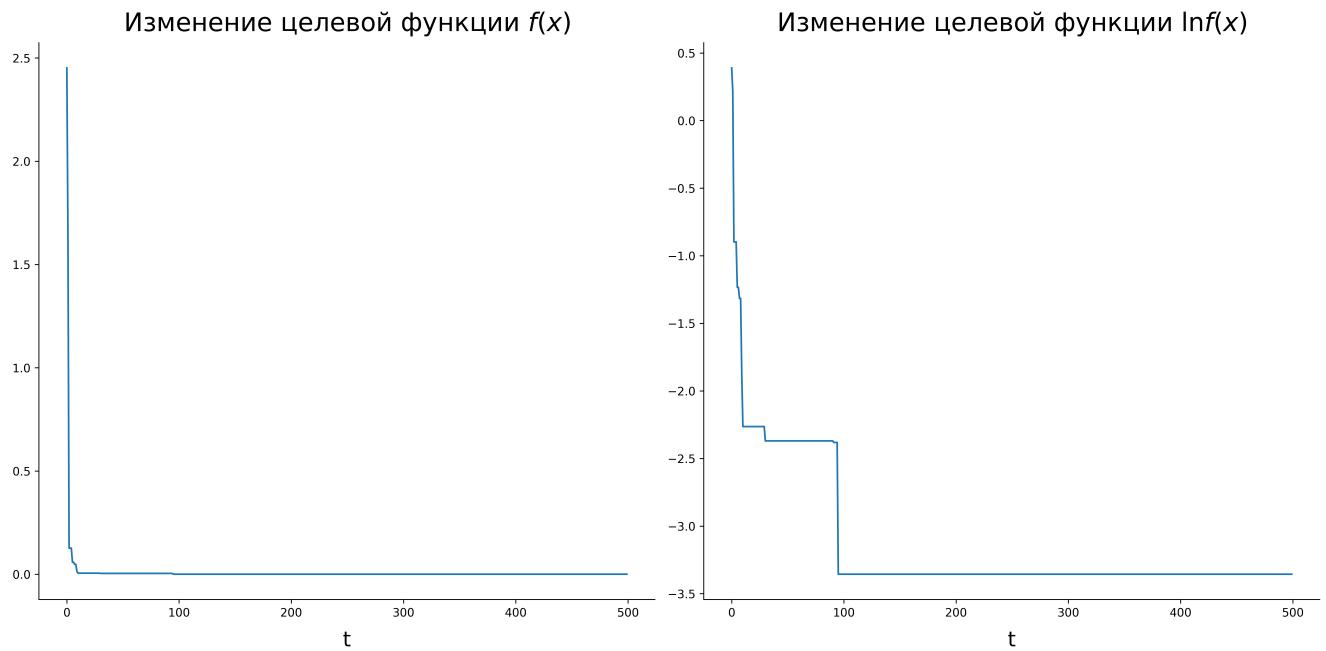


Рисунок 3.4 — Оптимизационный процесс.

Теперь реализуем наш алгоритм.

```
In: np.random.seed(13)
optimizer = GeneticAlgorithm(k=200, ell=1200, mutation_rate=0.2,
                             max_iter=500)
optimizer.search_global(search_space=[(-10, 10), (-10, 10)],
                        func=ackley_func)

Out: ##0216272: chromosome = (8.136619386361899e-05; -0.000132699205336);
      target_value = 0.0004406188188861293
```

3.3. Задача коммивояжера

Вернемся к задаче коммивояжера (TSP), рассмотренной нами ранее в разделе 1.4. Вспомним, что метод отжига — хоть и дал вполне приемлемый результат, — не смог найти самый оптимальный путь. Решим данную проблему посредством генетического алгоритма. Ко всему прочему, усложним TSP: будем искать оптимальный маршрут для 52 городов.

Определим класс City, моделирующий город.

```
In: class City:
    def __init__(self, coord, number):
        self.x = coord[0]
        self.y = coord[1]
        self.name = letters[number]

    def distance(self, city):
        return np.sqrt((self.x - city.x) ** 2 + (self.y - city.y) ** 2)

    def __repr__(self):
        return f'{self.name}: ({str(self.x)}; {self.y})'

part_1 = [f'{chr(i)}' for i in range(65, 65 + 26)]
part_2 = [f'{chr(i)}*' for i in range(65, 65 + 26)]
letters = part_1 + part_2
```

Создадим изначальный маршрут.

```
In: def map_city(cities_num, cities_range):
    return [City(np.random.randint(1, cities_range, size=2), j)
            for j in range(cities_num)]

In: np.random.seed(13)
not_closed_road = map_city(52, 2000)
closed_road = not_closed_road + [not_closed_road[0]]
```

Зададим нашу целевую функцию.

```
In: def total_distance(path, cities_lst):
    dist_lst = []
    for j in range(len(path) - 1):
        dist_lst.append(cities_lst[j].distance(cities_lst[j + 1]))
    dist_lst.append(cities_lst[j + 1].distance(cities_lst[0]))
    return sum(dist_lst)

In: path = [city.name for city in closed_road]
route_dist = total_distance(path=path, cities_lst=closed_road)
```

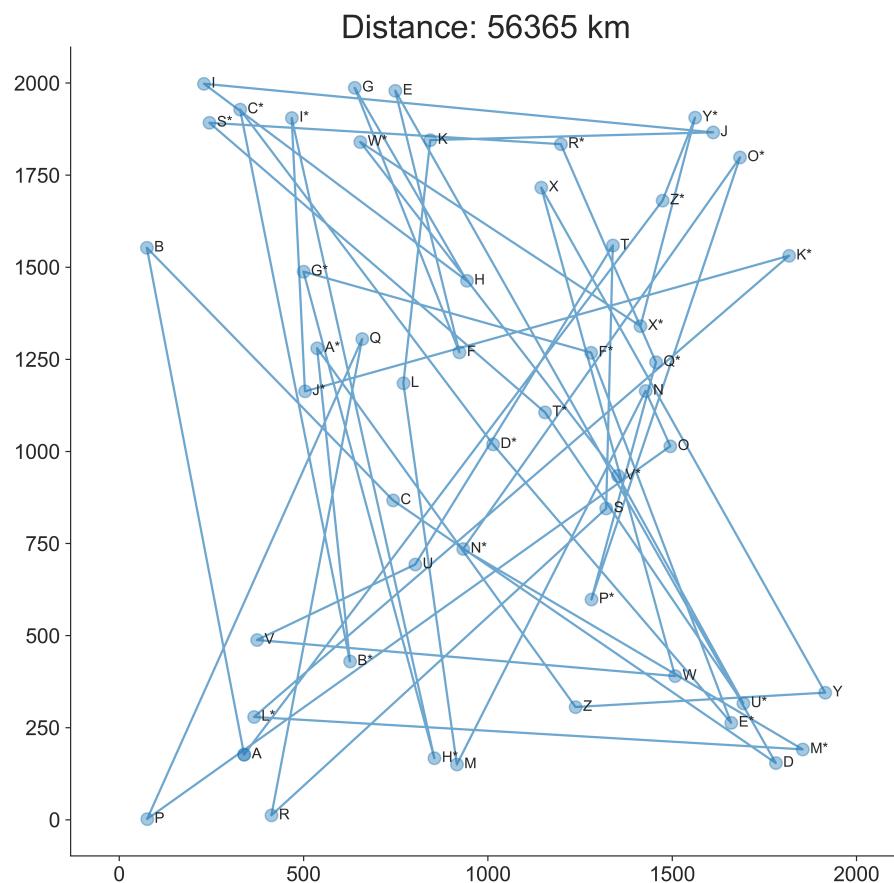


Рисунок 3.5 — Изначальный маршрут из 52-х городов.

Определим класс Individual для моделирования особи.

```
In: class Individual:
    def __init__(self, cities_lst, id_of_individual):
        self.map = random.sample(cities_lst[:-1],
                                  len(cities_lst[:-1]))
        self.path = [city.name for city in self.map]
        self.id_of_individual = id_of_individual
        self.dist = None
        self.fitness = None
```

```

def __repr__(self):
    id = self.id_of_individual
    return f'Individual {id}: Distance = {int(round(self.dist))} km'

```

Теперь определим генетический алгоритм.

```

In: class GeneticAlgorithm:
    def __init__(self, ell, k, mutation_rate, initial_map, max_iter):
        """
        PARAMETERS:
        ell --- количество особей в поколении
        k --- количество особей для размножения
        mutation_rate --- коэффициент мутации
        max_iter --- максимальное количество новых поколений
        """

        self.ell = ell
        self.k = k
        self.mutation_rate = mutation_rate
        self.initial_map = initial_map
        self.max_iter = max_iter

        self.best_individuals = None # k наиболее приспособленных особей
        self.fittest_individual = None # самая приспособленная особь
        self.population = None

    def search_best_path(self):
        # создаем первое поколение
        self.population = self.create_population()

        # проходим этапы эволюции
        for i in tqdm(range(self.max_iter)):
            # оцениваем приспособленность наших особей
            self.evaluate_population()
            # отбираем k наиболее наиболее приспособленных особей
            self.selection()

            # формируем новые поколения
            # скрещиваем особи
            for idx in range(self.k, self.ell):
                # случайно выбираем одну из наилучших особей
                agent = np.random.choice(self.best_individuals)
                select_fitted_individual = agent
                # создаем потомка и заменяем им старую особь
                offspring = self.crossover(select_fitted_individual,
                                            self.population[idx])

```

```

        self.population[idx].map = offspring
    # мутируем все особи, кроме самой приспособленной
    for individual in self.population[1:]:
        self.mutate(individual)
    return self.fittest_individual

def create_population(self):
    map = self.initial_map
    return [Individual(self.map, i + 1) for i in range(self.ell)]

def evaluate_population(self):
    F = []

    for indiv in self.population:
        indiv.dist = total_distance(indiv.path + [indiv.path[0]],
                                     indiv.map + [indiv.map[0]])
        F.append(indiv.dist)

    for indiv in self.population:
        indiv.fitness = self.normalize(indiv.dist,
                                        min(F), max(F))

def normalize(self, z, F_best, F_worst):
    """
    INPUT:
    z --- масштабируемое значение
    F_best --- лучшее значение целевой функции
    F_worst --- худшее значение целевой функции
    """
    return (z - F_worst) / (F_best - F_worst)

def crossover(self, parent_fitted, parent_random):
    """
    Operator скрещивания

    INPUT:
    parent_fitted --- одна из самых приспособленных особей
    parent_random --- случайная особь
    """
    genes_1 = int(np.random.uniform(0, 1) * len(parent_fitted.path))
    genes_2 = int(np.random.uniform(0, 1) * len(parent_fitted.path))

    start = min(genes_1, genes_2)
    end = max(genes_1, genes_2)

```

```

        xi = np.random.uniform(0, 1)
        if (end - start > self.ell / 2) and (xi < parent_fitted.fitness):
            child_1 = parent_fitted.map[start:end]
            child_2 = [i for i in parent_random.map if i not in child_1]
        else:
            child_1 = parent_random.map[start:end]
            child_2 = [i for o in parent_fitted.map if i not in child_1]

    return child_1 + child_2

def mutate(self, individual):
    """
    Оператор мутации

    INPUT:
    individual --- особь, подвергающаяся мутации
    """
    for j in range(len(individual.path)):
        if np.random.uniform(0, 1) < self.mutation_rate:
            num = np.random.randint(len(individual.path))
            a, b = individual.map[j], individual.map[num]
            individual.map[num], individual.map[j] = a, b

def selection(self):
    """
    Оператор отбора
    """
    self.population = sorted(self.population,
                             key=lambda x: x.fitness, reverse=True)
    self.best_individuals = self.population[:self.k]
    self.fittest_individual = self.population[0]

```

Реализуем его.

```

In: optimizer = GeneticAlgorithm(ell=1500, k=300, mutation_rate=0.001,
                                 initial_map=closed_road, max_iter=1500)
    optimizer.search_best_path()

Out: Individual 658: Distance = 11486 km

```

Из рисунка 3.7 можно увидеть, что генетический алгоритм справился с TSP и нашел самый оптимальный маршрут.

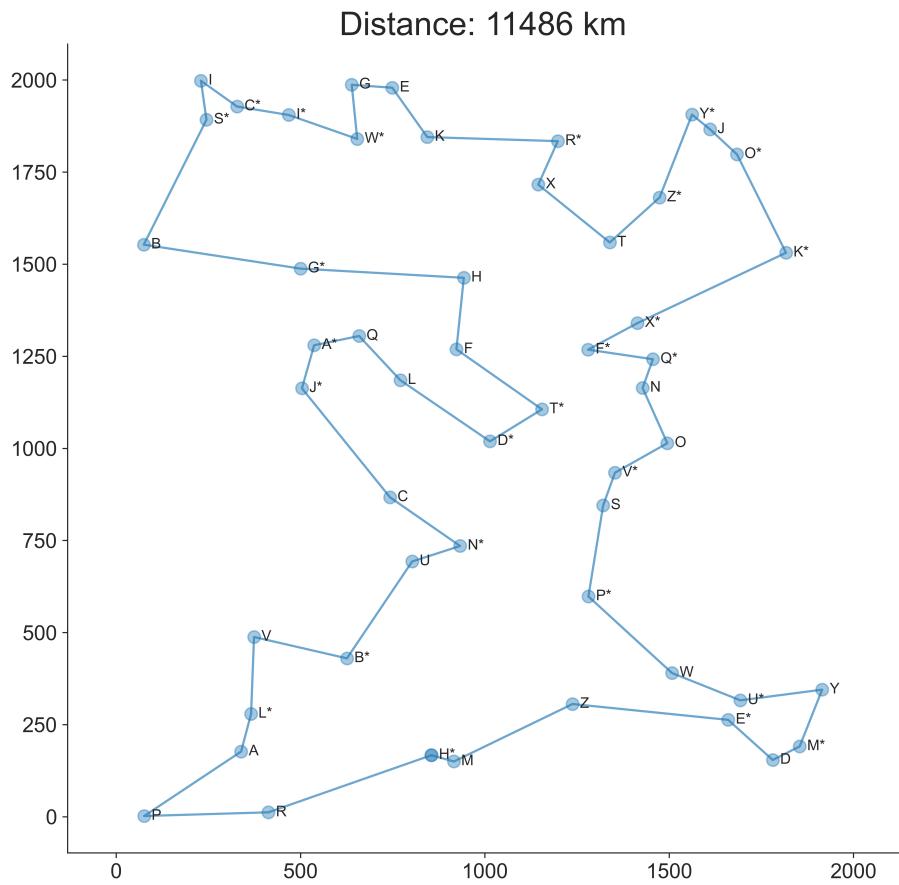


Рисунок 3.6 — Оптимальный маршрут из 52-х городов.

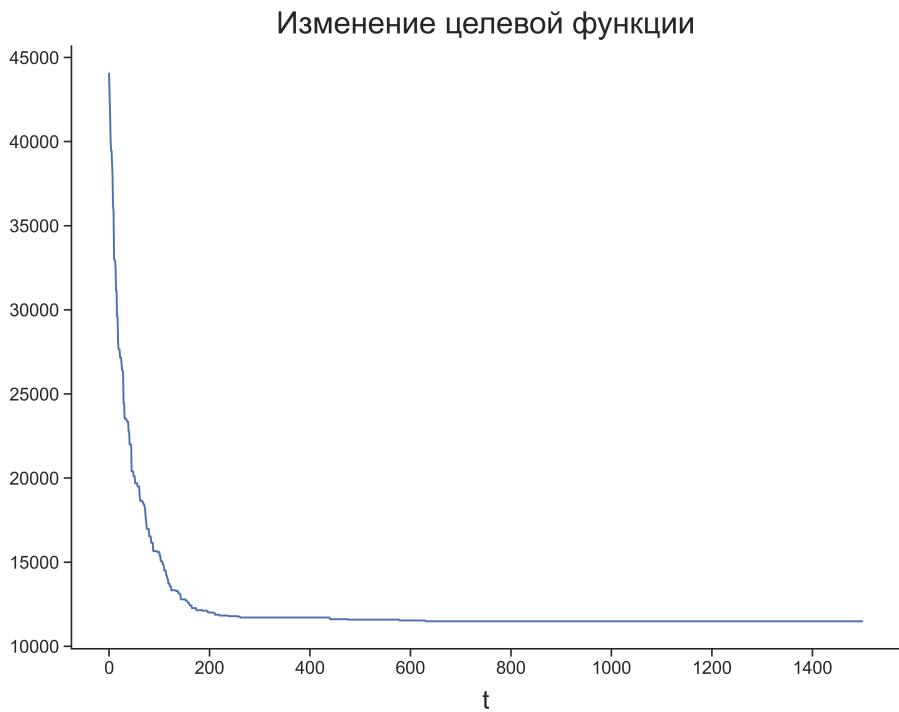


Рисунок 3.7 — Оптимизационный процесс.

3.4. Вывод

Рассмотрев генетический алгоритм, выявим его плюсы и минусы. РАСПИСАТЬ ЕГО МИНУСЫ ПРИМНОГО СКРЕМАЛЬНОСТИ И МЕДЛЕННОЙ РАБОТЕ (ПРИВЕСТИ ПРИМЕРЫ КОТОРЫЕ РАССМАТРИВАЛИ) СРАВНИТЬ С ТОЧНОСТЬЮ PSO

Таблица 3.1 — Генетический алгоритм

Преимущества	Недостатки
1. Не требует дифференцируемости и непрерывности исследуемой функции	1. Сложно найти глобальный оптимум с высокой заданной точностью
2. Используется для широкого класса задач	2. Плохо работает для многоэкстремальных функций
3. Не застревает в локальных минимумах, если их не много	3. Достаточно медленная реализация
4. Имеет простую реализацию	

Заключение

Основные результаты работы заключаются в следующем.

Список литературы

- [1] *Воронцов К. В.* (2010) Курс по машинному обучению. Критерии выбора моделей и методы отбора признаков. // Сайт machinelearning.ru. URL: <http://machinelearning.ru> (дата обращения: 11.05.2020)
- [2] *Лопатин А. С.* (2005) Стохастическая оптимизация в информатике. Метод отжига. // Сайт math.spbu.ru. URL: <https://www.math.spbu.ru/user/gran/optstoch.htm> (дата обращения: 20.01.2020)
- [3] *Панченко Т. В.* (2007) Генетические алгоритмы: учебно-методическое пособие. // Астрахань: Издательский дом «Астраханский университет».
- [4] *Шамин Р. В.* (2019) Машинное обучение в задачах экономики. // Москва: Грин Принт.
- [5] *Шамин Р. В.* (2019) Практическое руководство по машинному обучению. // Москва: Научный канал.
- [6] *Nedjah N., Mourelle L.* (2006) Swarm Intelligent Systems. // Berlin, Heidelberg: Springer Berlin Heidelberg.
- [7] *Pedersen M. E. H.* (2010) Tuning and Simplifying Heuristical Optimization. Thesis for the degree of Doctor of Philosophy. // University of Southampton.
- [8] Petrowski A., Ben-Hamida S. (2017) Evolutionary Algorithms. // Hoboken: New Jersey.
- [9] *Weise T.* (2009) Global Optimization Algorithms — Theory and Application. // Self-Published.