федеральное государственное бюджетное образовательное
учреждение высшего образования
«Национальный исследовательский университет «МЭИ»

Институт информационных и вычислительных технологий
Кафедра управления и интеллектуальных технологий

Отчёт по лабораторной работе №2
По дисциплине «Нейро-нечёткие технологии в задачах управления»
«Основы генетических алгоритмов»

Выполнили студенты: Михайловский М., Томчук В.
Группа: А-03-21
Бригада: 1
Проверил: Косинский М. Ю.

Москва 2024

Содержание

1 Постановка задачи

Дана функция $f(x) = 2x^2 + 1$, необходимо найти её максимум с точностью до целого значения $x \in [0, 15]$ с применением генетического алгоритма. График целевой функции $f(x)$ представлен на рис. 1.1.

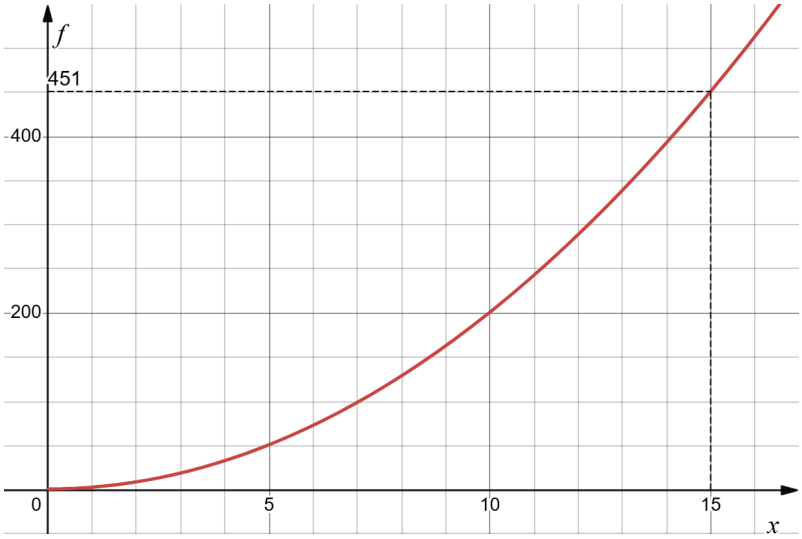

Рис. 1.1. График целевой функции

В рамках работы генетического алгоритма ищется такой генетический код $x_2 \equiv x$, такой что, функция приспособленности для него максимальна: $f(x) \to \max$. Здесь x_2 является бинарным представлением $x \in \mathbb{R}$.

2 Описание генетического алгоритма

Алгоритм имеет следующий вид (изначальная популяция особей уже выбрана и представлена в бинарном коде, изначальная приспособленность рассчитана):

1. Селекция особей для скрещивания;
2. Мутация особей;
3. Получение новой популяции в результате скрещивания;
4. Расчёт приспособленности полученных особей;
5. Проверка критерия остановки.

2.1 Селекция

В ходе работы было реализовано два метода селекции: классический и ранговый. Они определяют то, как из популяции размером N выбираются особи для скрещивания.

Классический. Для каждой i-ой особи рассчитывается относительная приспособленность:

$$f_{\text{отн}}(x_i) = \frac{f(x_i)}{\sum_{k=1}^{N} f(x_k)} \tag{1}$$

Разыгрывается так называемая «рулетка». По сути с помощью случайной величины $\xi \sim R(0,1)$ выбирается особь для скрещивания. Вся область определения случайной величины ξ делится и ставится в соответствие каждой особи, при чем величина сопоставляемого интервала соответствует относительной приспособленности $f_{\text{отн}}(x_i)$ этой особи.

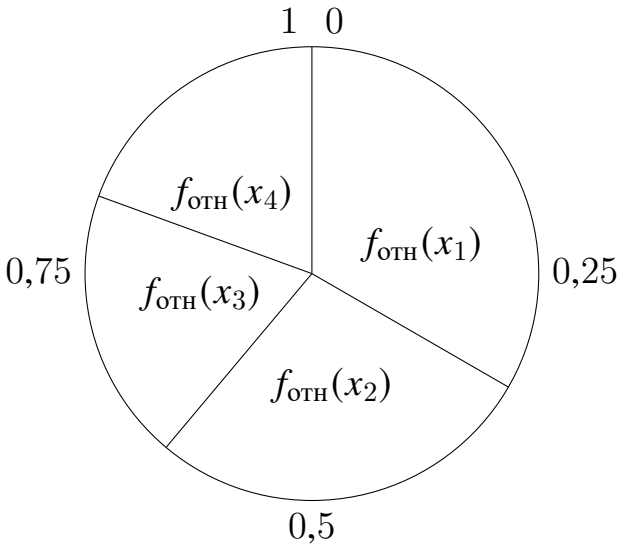

Рис. 2.1. Распределение области определения ξ в соответствии с относительными приспособленностями особей

Ранговый. Для каждой особи рассчитывается относительная приспособленность в соответствии с формулой (1). Всей популяции в соответствии в зависимости от величины относительной приспособленности присуждаются ранги, от наибольшего, который соответствует наиболее приспособленной особи, к наименьшему.

Для скрещивания выбираются особи, в соответствии с заданным требуемым распределением рангов, например три особи 1 ранга, две особи 2 ранга и одна особь 3 ранга.

2.2 Мутация особей

В самом алгоритме задаётся вероятность мутации. Если мутация происходит, то в генетическом коде особи случайно выбранный ген изменятся. Это равносильно отрицанию одного из битов бинарного кода:

$$101110 \rightarrow 100110$$

Такой стохастический элемент в алгоритме помогает избежать локальных оптимумов создаваемых конкретной текущей популяцией, не имеющих нужных генов для достижения глобального оптимума.

2.3 Скрещивание

В скрещивании участвуют две особи. Выбирается случайная точка скрещивания: начиная с неё, родители обмениваются последующим генетическим кодом, получая двух новых особей:

$$\begin{array}{lccccccc} \text{1 родитель:} & 11111 & & 11\ 111 & & 11\ 000 & & 11000 \\ & & \Rightarrow & \updownarrow & \Rightarrow & & \Rightarrow & \\ \text{2 родитель:} & 00000 & & 00\ 000 & & 00\ 111 & & 00111 \end{array}$$

3 Реализованная программа

Программа была написана на языке *python 3.11.3* с использованием библиотеки *numpy*. Программа состоит из двух модулей:

- **main.py** – основной модуль, где реализован собственно генетический алгоритм;
- **Gen.py** – модуль, в котором реализован класс, для работы с генетическими кодами, то есть рассматриваемыми особями.

Листинг этих модулей приведён в приложении. Примеры работы программы с классической и ранговой селекцией приведены ниже:

Листинг 1. Пример работы программы с классической селекцией

```
1 Классический метод селекции. Вероятность мутации 0.2
2 {Начало вывода программы}
3 Поколение    1: 1100 1110 1001 0101     Лучший ген 1110. Лучшая приспособленность
  ↪  393
```

```
                  32%  44%  18%   6%
         Выбраны для скрещивания: 1110 1110 1100 1110
         Гены после мутации: 1000 1000 1100 1000
         Результат скрещивания: 1000 1000 1000 1100

Поколение    2: 1000 1000 1000 1100     Лучший ген 1110. Лучшая приспособленность
 ↪  393
                  19%  19%  19%  43%
         Выбраны для скрещивания: 1100 1000 1100 1000
         Гены после мутации: 1100 1000 1100 1000
         Результат скрещивания: 1000 1100 1000 1100

Поколение    3: 1000 1100 1000 1100     Лучший ген 1110. Лучшая приспособленность
 ↪  393
                  15%  35%  15%  35%
         Выбраны для скрещивания: 1000 1000 1100 1000
         Гены после мутации: 1000 1000 1110 1100
         Результат скрещивания: 1000 1000 1100 1110

Поколение    4: 1000 1000 1100 1110     Лучший ген 1110. Лучшая приспособленность
 ↪  393
                  14%  14%  31%  42%
         Выбраны для скрещивания: 1000 1000 1100 1110
         Гены после мутации: 1000 1000 1100 1111
         Результат скрещивания: 1000 1000 1111 1100
         Ген 1111 лучше 1111, потому что 451 > 451

Поколение    5: 1000 1000 1111 1100     Лучший ген 1111. Лучшая приспособленность
 ↪  451
                  13%  13%  45%  29%
```

Листинг 2. Пример работы программы с ранговой селекцией

```
Ранговый метод селекции. Вероятность мутации 0.2
{Начало вывода программы}
Поколение    1: 0000 0011 1001 1001     Лучший ген 1001. Лучшая приспособленность
 ↪  163
                   0%   5%  47%  47%
         Выбраны для скрещивания: 1001 1001 1001 0011
         Гены после мутации: 1101 1101 1001 0011
         Результат скрещивания: 1101 1101 1011 0001
         Ген 1101 лучше 1101, потому что 339 > 339

Поколение    2: 1101 1101 1011 0001     Лучший ген 1101. Лучшая приспособленность
 ↪  339
                  37%  37%  26%   0%
         Выбраны для скрещивания: 1101 1101 1101 1011
         Гены после мутации: 1101 1101 1101 1011
         Результат скрещивания: 1101 1101 1111 1001
         Ген 1111 лучше 1111, потому что 451 > 451

Поколение    3: 1101 1101 1111 1001     Лучший ген 1111. Лучшая приспособленность
 ↪  451
```

```
                  26%  26%  35%  13%
        Выбраны для скрещивания: 1111 1111 1101 1101
        Гены после мутации: 1101 1101 1111 1101
        Результат скрещивания: 1101 1101 1111 1101

Поколение    4: 1101 1101 1111 1101     Лучший ген 1111. Лучшая приспособленность
 ↪  451
                  23%  23%  31%  23%
        Выбраны для скрещивания: 1111 1111 1101 1101
        Гены после мутации: 1101 1101 1101 1101
        Результат скрещивания: 1101 1101 1101 1101

Поколение    5: 1101 1101 1101 1101     Лучший ген 1111. Лучшая приспособленность
 ↪  451
                  25%  25%  25%  25%
```

Как видно, в обоих случаях был достигнут глобальный оптимум исследуемой задачи. В общем случае, за 4 шага, конечно, не всегда будет достигаться такой результат. Но принцип работы алгоритма здесь явно виден, и наблюдается общее увеличение приспособленности популяции от поколения к поколению.

Приложение А. Код программы

Листинг 3. main.py

```python
from Gen import Gen
import random
import numpy as np

def f(x):
    return 2 * (x ** 2) + 1

def selection(gens, relative_adapt, generation_size, method: str):
    valid_methods = {
        'classic',
        'rank'
    }
    if method not in valid_methods:
        raise ValueError(f'Указан несуществующий метод {method}. Возможные
         ↪  варианты: {repr(valid_methods)}')

    if method == 'classic':
        result = [ random.choices(gens, relative_adapt, k=1)[0] for _ in
         ↪  range(generation_size) ]

    select_distribution = {
        4: [2, 1, 1],
        6: [3, 2, 1],
        8: [3, 2, 2, 1]
```

```
    }
    if method == 'rank':
        if generation_size not in select_distribution:
            raise ValueError(f'Не определено распределение для выбора по ранговому
            ↪  методу для размера выборки {generation_size}')

        gens_ranked = np.array([gens, relative_adapt])
        gens_ranked = gens_ranked[:, gens_ranked[1,:].argsort()[::-1] ]

        result = []
        select_counts = select_distribution[generation_size]
        for i in range(len(select_counts)):

            result += [gens_ranked[0,i], ] * select_counts[i]

    return result


def generate_population(x_min, x_max, count, length):
    population = []
    sum_adapt = 0
    for _ in range(count):
        x = random.randint(x_min, x_max)
        gen = Gen(length, x=x)
        adapt = f(x)
        sum_adapt += adapt
        info = {
            'gen': gen,
            'adapt': adapt
        }
        population.append(info)
    return (population, sum_adapt)

def mutate_population(gens, p_mutation):
    for i, gen in enumerate(gens):
        if random.random() < p_mutation:
            gen.mutate()

def print_stats(gens, best_gen, best_adapt, relative_adapt, step,
↪  descriptive_log):
    print(f'Поколение {step:>4}: ', end='')
    [ print(gen, end=' ') for gen in gens ]
    print(f'\tЛучший ген {best_gen}. Лучшая приспособленность {best_adapt}')
    if descriptive_log:
        print('\t\t', end='')
        [ print(f'{r_adapt:>3.0f}%', end=' ') for r_adapt in relative_adapt ]
        print()

def print_gens(text, gens):
    print(f'{text}', end='')
    [ print(gen, end=' ') for gen in gens ]
    print()

def main():
```

```
75      #Общие параметры
76      p_mutation = 0.2
77      gen_length = 4
78      generation_size = 4
79      selection_method = 'classic' #classic или rank
80      max_steps = 4
81      descriptive_log = True
82
83      #Параметры генерации
84      x_min = 0
85      x_max = 15
86
87      population, sum_adapt = generate_population(x_min, x_max, generation_size,
        ↪  gen_length)
88      relative_adapt = [ 100*gen_info['adapt']/sum_adapt for gen_info in population
        ↪  ]
89      gens = [ gen_info['gen'] for gen_info in population ]
90
91      max_ind = relative_adapt.index(max(relative_adapt))
92      best_gen = str(gens[max_ind])
93      best_adapt = gens[max_ind].apply(f)
94
95      step = 1
96      print_stats(gens, best_gen, best_adapt, relative_adapt, 1, descriptive_log)
97      while step <= max_steps:
98          #Селекция
99          gens_to_cross = selection(gens, relative_adapt, generation_size,
            ↪  selection_method)
100         if descriptive_log:
101             print_gens('\tВыбраны для скрещивания: ', gens_to_cross)
102
103         #Мутация
104         mutate_population(gens_to_cross, p_mutation)
105         if descriptive_log:
106             print_gens('\tГены после мутации: ', gens_to_cross)
107
108         #Скрещивание
109         new_gens = []
110         sum_adapt = 0
111         for i in range(int(generation_size / 2)):
112             gens_to_add = gens_to_cross[2*i].cross(gens_to_cross[2*i+1])
113             sum_adapt += gens_to_add[0].apply(f) + gens_to_add[1].apply(f)
114             new_gens += gens_to_add
115         gens = new_gens
116         relative_adapt = [ 100*gen.apply(f)/sum_adapt for gen in gens ]
117         if descriptive_log:
118             print_gens('\tРезультат скрещивания: ', gens)
119
120         #Выбор лучшего
121         max_ind = relative_adapt.index(max(relative_adapt))
122         max_gen = gens[max_ind]
123         max_adapt = gens[max_ind].apply(f)
124
125         if max_adapt > best_adapt:
```

```
126             best_gen = str(max_gen)
127             best_adapt = max_adapt
128 
129             if descriptive_log:
130                 print(f'\tГен {max_gen} лучше {best_gen}, потому что {max_adapt} >
                    ↪ {best_adapt}')
131 
132         #Печать поколения
133         if descriptive_log:
134             print()
135         print_stats(gens, best_gen, best_adapt, relative_adapt, step+1,
            ↪ descriptive_log)
136 
137         step += 1
138 
139 
140 if __name__ == '__main__':
141     main()
```

Листинг 4. Gen.py

```
1  import random
2  
3  class Gen():
4      @staticmethod
5      def intToBin(x: int):
6          return bin(x)[2:]
7  
8      @staticmethod
9      def binToInt(code: str):
10         return int(code, 2)
11 
12     def __init__(self, length, code: list = None, x = None, binarizator = None):
13         if binarizator:
14             self.binarizator = binarizator
15         else:
16             self.binarizator = Gen.intToBin
17 
18         if x is not None:
19             #Создание из изначального целочисленного объекта
20             code = self.binarizator(x)
21             if len(code) > length:
22                 raise ValueError(f'Длина полученного кода ({len(code)}) больше
                    ↪ заданной длины хромосом ({length}).')
23 
24             if len(code) < length:
25                 code = '0' * (length - len(code)) + code
26             code = [ int(char) for char in code ]
27             self.code = code
28         elif code is not None:
29             #Создание из изначального кода
30             self.code = code
```

```
31
32          if len(code) != length:
33                  raise ValueError(f'Длина полученного кода ({len(code)}) больше
                    ↪  заданной длины хромосом ({length}).')
34
35          self.code_length = length
36
37      def __str__(self):
38          res = ''
39          for elem in self.code:
40              res += str(elem)
41          return res
42
43      def mutate(self):
44          place = random.randint(0, self.code_length-1)
45          self.code[place] = int(not self.code[place])
46
47      def cross(self, otherGen: 'Gen') -> list:
48          #Скрещивание
49          if self.code_length != otherGen.code_length:
50              raise ValueError('Длины кодов хромосом не совпадают')
51
52          point = random.randint(1, self.code_length-1)
53          code1 = self.code[:point] + otherGen.code[point:]
54          gen1 = Gen(self.code_length, code=code1)
55          code2 = otherGen.code[:point] + self.code[point:]
56          gen2 = Gen(self.code_length, code=code2)
57          return [gen1, gen2]
58
59      def apply(self, func):
60          value = Gen.binToInt(str(self))
61          return func(value)
62
```