ZAD 1

Proszę zaproponować swoją implementację algorytmu genetycznego w celu znalezienia maksimum funkcji

$$f(x) = \cos(80x + 0.3) + 3x^{-0.9} - 2$$

w przedziale $[0.01, 1]$. Dla $x = 0$ proszę przyjąć $f(x) = 0$ (choć nie powinno być używane w zadaniu). Proszę porównać działanie algorytmu:

- Dla kodowania w naturalnym kodzie binarnym i w kodzie Graya
- Dla szansy mutacji wynoszącej 0, 0.1, 0.5 i 1.0
- Dla selekcji ruletkowej i dla selekcji progowej. W selekcji progowej dzielimy populację na dwie grupy: na $\gamma\%$ najlepszych i na pozostałych. Osobniki w grupie $\gamma\%$ najlepszych mają równą szansę na reprodukcję, pozostałe mają zerową szansę na reprodukcję. Proszę sprawdzić wyniki dla $\gamma = 30$ i $\gamma = 60$.

Za każdym razem proszę podać średnie wyniki dla 10 wywołań algorytmu i przedstawić przykładowe przebiegi algorytmu na wykresach (dla jednego z wywołań).

In [139…

```
import random
import math
import matplotlib.pyplot as plt

class Solution:
    def __init__(self, randomize_genes=False):
        self.genes = [0] * 16
        if randomize_genes:
            self.genes = [random.randint(0, 1) for _ in range(16)]

    def get_number(self, kind = 'bin'):
        number = 0
        if kind == 'bin':
          int_val = int("".join(str(x) for x in self.genes), 2)
          number = 0.01 + (int_val / (2**16 - 1)) * (1.0 - 0.01)
        elif kind == 'gray':
          binary_code = [self.genes[0]]
          for i in range(1, len(self.genes)):
              binary_code.append(self.genes[i] ^ binary_code[i - 1])
          int_val = int("".join(str(x) for x in binary_code), 2)
          number = 0.01 + (int_val / (2**16 - 1)) * (1.0 - 0.01)
        return number

    def get_adaptation(self, kind = 'bin'):
        number = self.get_number(kind)
```

```
        return math.cos(80 * number + 0.3) + 3 * number ** (-0.9) -

    def crossover(self, other_solution):
        cut_position = random.randint(0,15)
        new_solution = Solution()
        new_solution.genes[0:cut_position] = self.genes[0:cut_posit
        new_solution.genes[cut_position:] = other_solution.genes[cu
        return new_solution

    def mutation(self):
        mutation_position = random.randint(0, 15)
        if self.genes[mutation_position] == 1:
          self.genes[mutation_position] = 0
        else:
          self.genes[mutation_position] = 1
```

In [140…

```
def print_dashes():
  return '-'*30
```

In [141…

```
population_size = 50
iterations = 50

def run_algorithm(coding_type, mutation_chance, selection_type, pri
  population = [Solution(randomize_genes = True) for i in range(pop
  best_solution = Solution()
  best_solution_adaptation = 0.
  best_iteration_found = 0

  avgs = []
  bests_local = []
  bests_global = []

  for iteration in range(iterations):
    adaptations = [p.get_adaptation(kind = coding_type) for p in po
    local_best_solution = population[adaptations.index(max(adaptati
    if local_best_solution.get_adaptation(kind = coding_type) > bes
      best_solution = local_best_solution
      best_solution_adaptation = local_best_solution.get_adaptation
      best_iteration_found = iteration

    avgs.append(sum(adaptations) / len(adaptations))
    bests_local.append(max(adaptations))
    bests_global.append(best_solution_adaptation)

    if selection_type == 'roulette':
      roulette_wheel = adaptations
      for i in range(len(roulette_wheel)):
        roulette_wheel[i] -= min(adaptations)
        roulette_wheel[i] /= (max(adaptations)-min(adaptations))

      parents = [random.choices(population, weights=roulette_wheel,
      children = [p[0].crossover(p[1]) for p in parents]

    else:
      threshold_value = round(int(selection_type[-2:])/100*populati
```

```
      best_candidates = sorted(population, key=lambda x: x.get_adap
      parents = [random.choices(best_candidates, k=2) for i in rang
      children = [p[0].crossover(p[1]) for p in parents]

    for c in children:
        if random.random() < mutation_chance:
            c.mutation()
    population = children

  # modyfikujemy ostateczny najlepszy wynik
  adaptations = [p.get_adaptation(kind = coding_type) for p in popu
  local_best_solution = population[adaptations.index(max(adaptation
  if local_best_solution.get_adaptation(kind = coding_type) > best_
    best_solution = local_best_solution
    best_solution_adaptation = local_best_solution.get_adaptation(k

  if print_details:
    print('---')
    print('Best solution: ', best_solution.genes, ' = ', best_solut
    print('Found in iteration: ', best_iteration_found)
    print('Largest function value found: ', best_solution_adaptatio

  if show_plots:
    fig, axs = plt.subplots(1, 3, figsize=(15, 5))
    axs[0].plot(avgs)
    axs[0].set_title("Średnia wartość funkcji przystosowania w popu
    axs[1].plot(bests_local)
    axs[1].set_title("Przystosowanie najlepszego osobnika w iteracj
    axs[2].plot(bests_global)
    axs[2].set_title("Przystosowanie najlepszego osobnika globalneg
    plt.tight_layout()
    plt.show()

  return best_solution, best_solution_adaptation
```

In [142…

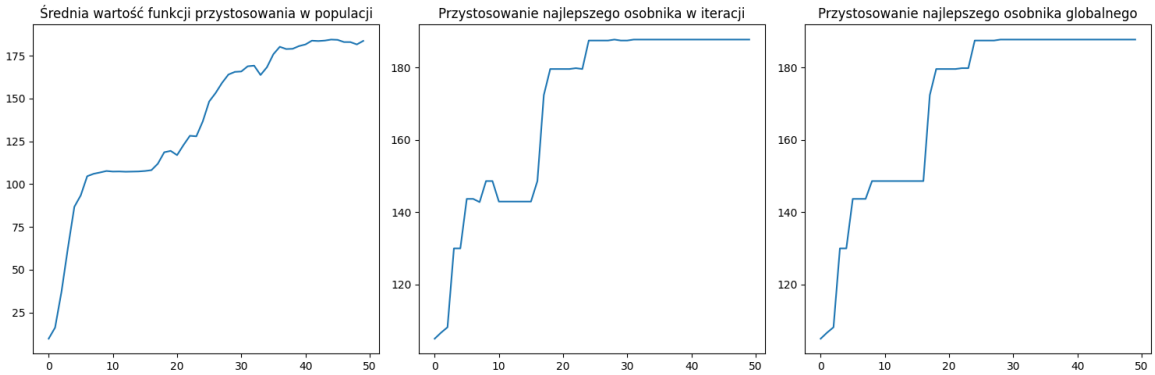
```
import numpy as np

mutation_chances = [0, 0.1, 0.5, 1.0]
codings = ['bin', 'gray']
selection_types = ['roulette', 'treshold_30', 'treshold_60']

for coding_type in codings:
  print(f'{print_dashes()} CODING TYPE: {coding_type} {print_dashes
  for mutation_chance in mutation_chances:
    print(f'{print_dashes()} MUTATION CHANCE: {mutation_chance} {pr
    for selection_type in selection_types:
      print(f'{print_dashes()} SELECTION TYPE: {selection_type} {pr
      best_solutions = []
      best_adaptations = []
      for i in range(10):
        print(f'RUN {i+1}')
        if i == 6:
          best_solution, best_adaptation = run_algorithm(coding_typ
        else:
          best_solution, best_adaptation = run_algorithm(coding_typ
        best_solutions.append(best_solution)
```

```
            best_adaptations.append(best_adaptation)
        avg_best_adaptation = np.mean(best_adaptations)
        avg_best_solution = sum(solution.get_number(kind=coding_type)
        print(f'Average best adaptation from 10 runs: {avg_best_adapt
        print(f'Average best solution: {avg_best_solution:.8f}')
        print()
```

```
------------------------------ CODING TYPE: bin ------------------------
----------
------------------------------ MUTATION CHANCE: 0 ----------------------
------------
------------------------------ SELECTION TYPE: roulette ----------------
----------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7
```




```
RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 173.777683
Average best solution: 0.01120096

------------------------------ SELECTION TYPE: treshold_30 -------------
--------------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7
```

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 159.958256
Average best solution: 0.04027782

------------------------------ SELECTION TYPE: treshold_60 ----------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 148.328831
Average best solution: 0.01394127

------------------------------ MUTATION CHANCE: 0.1 ----------------
----

------------------------------ SELECTION TYPE: roulette ------------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 186.286700
Average best solution: 0.01008762

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ SELECTION TYPE: treshold_60 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ MUTATION CHANCE: 0.5 ----------------
----
------------------------------ SELECTION TYPE: roulette ------------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.714994
Average best solution: 0.01000151

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ SELECTION TYPE: treshold_60 ----------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ MUTATION CHANCE: 1.0 ----------------
----

------------------------------ SELECTION TYPE: roulette ------------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.689188
Average best solution: 0.01000302

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ SELECTION TYPE: treshold_60 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.689188
Average best solution: 0.01000302

------------------------------ CODING TYPE: gray ------------------------------
------------------------------ MUTATION CHANCE: 0 ------------------------------
------------------------------ SELECTION TYPE: roulette ------------------------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

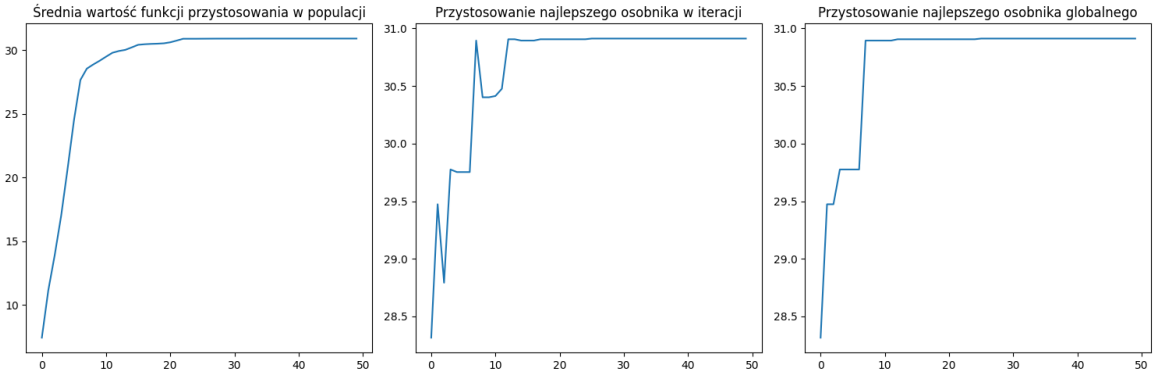
RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 147.587158
Average best solution: 0.01699126

------------------------------ SELECTION TYPE: treshold_30 ------------------------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 102.157619
Average best solution: 0.02826217

------------------------------ SELECTION TYPE: treshold_60 ----------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

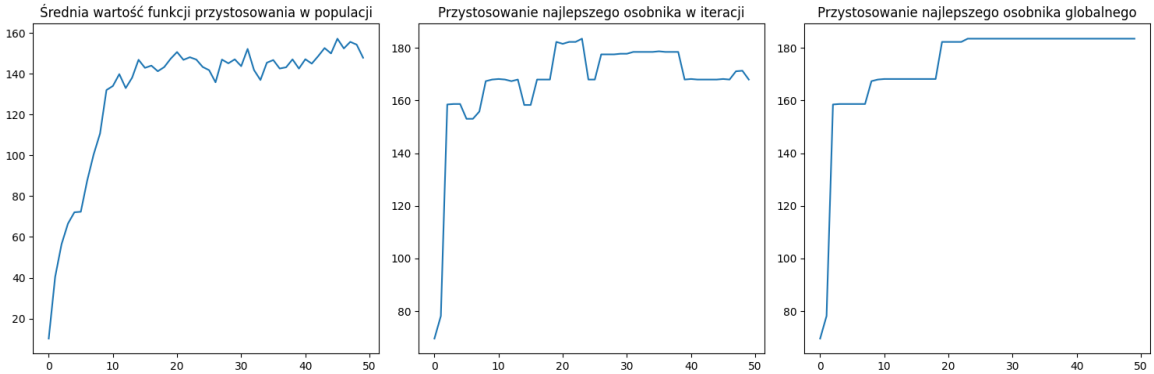


RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 154.764341
Average best solution: 0.01748222

------------------------------ MUTATION CHANCE: 0.1 ----------------
----

------------------------------ SELECTION TYPE: roulette ------------
----

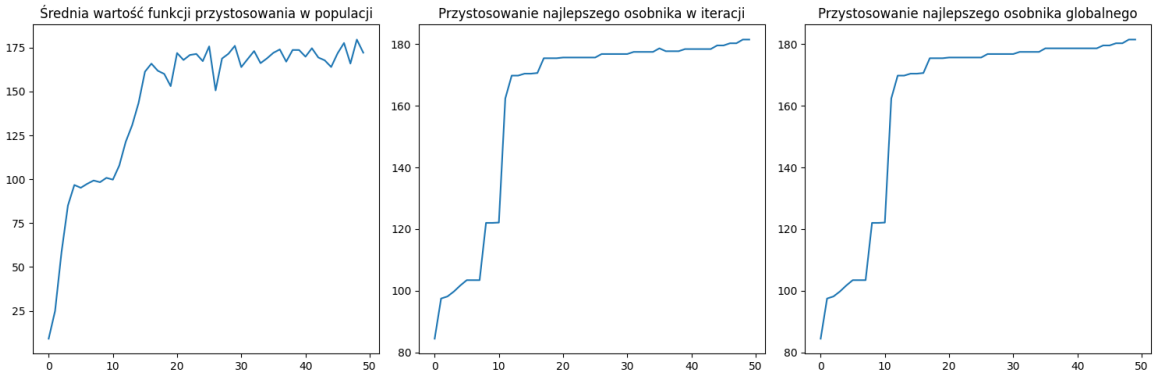
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7



RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 183.997394
Average best solution: 0.01022962

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7



RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.014195
Average best solution: 0.01004381

------------------------------ SELECTION TYPE: treshold_60 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 180.627483
Average best solution: 0.01053477

------------------------------ MUTATION CHANCE: 0.5 ----------------
----
------------------------------ SELECTION TYPE: roulette ------------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.663455
Average best solution: 0.01000453

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

```
RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ SELECTION TYPE: treshold_60 ---------
----------------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7
```


```
RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ MUTATION CHANCE: 1.0 ----------------
--------------
------------------------------ SELECTION TYPE: roulette ------------
------------------
RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7
```

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.637796
Average best solution: 0.01000604

------------------------------ SELECTION TYPE: treshold_30 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.740799
Average best solution: 0.01000000

------------------------------ SELECTION TYPE: treshold_60 ---------
----

RUN 1
RUN 2
RUN 3
RUN 4
RUN 5
RUN 6
RUN 7

```
RUN 8
RUN 9
RUN 10
Average best adaptation from 10 runs: 187.586477
Average best solution: 0.01000906
```

Przykładowe przebiegi algorytmu na wykresach dla jednego z wywołań dla każdej konfiguracji:

Wnioski

Kod binarny

Dla kodu binarnego widzimy polepszenie wyników względem prawdopodobieństwa mutacji 0 do pozostałych wartości prawdopodobieństwa. Dla prawdopodobieństwa 0, wszystkie typy selekcji dają gorszy wynik, szczególnie threshold_30 tutaj wyłania się z wynikiem najsłabszym. Przy wartości prawdopodobieństwa 0.1, selekcja roulette daje słabszy wynik od thresholdów, które tutaj już zwracają idealny wynik. Dla następnych wartości prawdopodobieństwa mutacji (0.5, 1), trzy typy selekcji działają porównywalnie, znajdując globalne maksimum prawidłowo.

Kod gray'a

W przypadku kodowania gray'a dostajemy gorsze wyniki ogólnie, dla każdego z przypadków. Również tutaj, jak w kodzie binarnym widzimy polepszenie wyników wraz ze wzrostem wartości prawdopodobieństwa mutacji, jednak tutaj ani roulette, ani threshold60, ani razu nie znajdują idealnie prawidłowej wartości. Zawsze są blisko niej. Inaczej jest z selekcją threshold30, która już dla mutacji 0.5 wzwyż zwraca prawidłowe maksimum oraz dla mutacji 0.1 zwraca bardzo zbliżony wynik do wyniku prawidłowego, gdzie wyniki roulette i threshold60 odbiegają jeszcze sporo od prawidłowego maksimum. To ciekawe, ponieważ tak samo jak dla kodu binarnego, threshold30 daje najgorszy wynik przy mutacji 0 względem innych typów selekcji. Jednak potem, dla większych wartości prawdopodobieństwa mutacji, dla kodu graya, jest optymalnym typem selekcji.

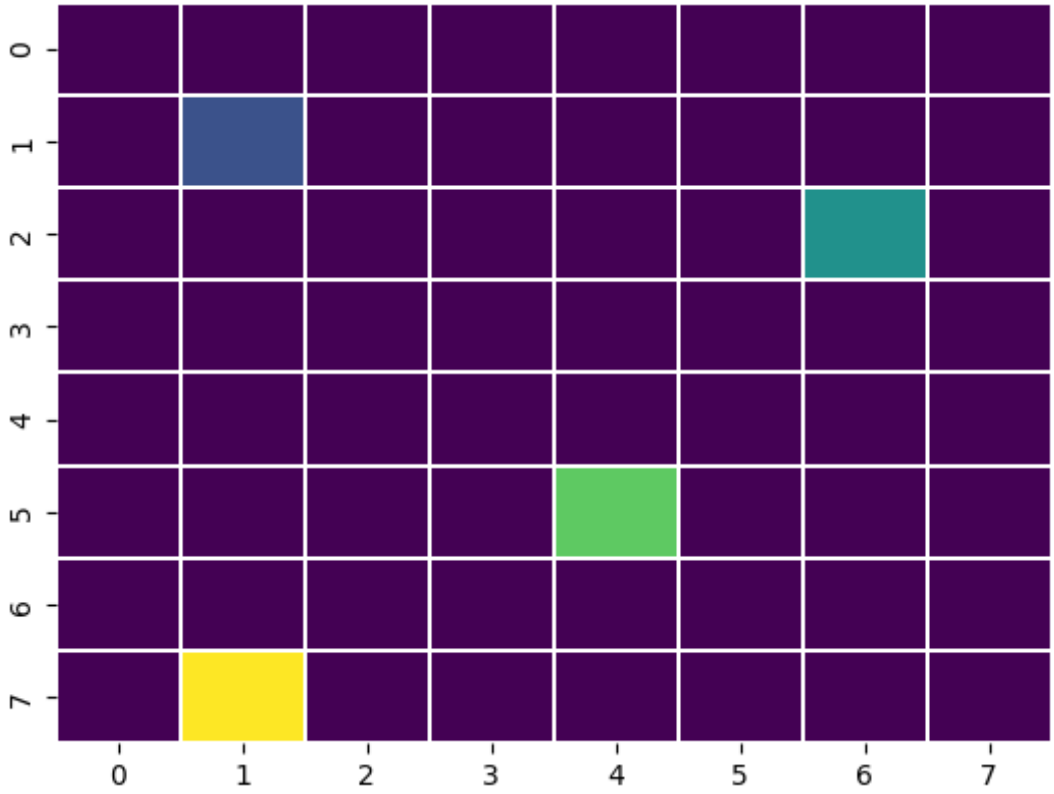
Podsumowanie

Wykorzystanie kodowania binarnego ogólnie polepsza wyniki. Wtedy trzy typy selekcji szybko znajdują prawidłowe maksimum globalne przy odpowiednio dużej wartości prawdopodobieństwa mutacji, gdy dla kodowania gray'a, tylko threshold30 zwraca prawidłowe maksimum. Jeżeli musiałabym wybrać kodowanie gray'a dla tego problemu, wykorzystałabym selekcję treshold30 i

mutację min 0.5. Dla takiego kodowania miałabym mniejsze pole wyboru rodzaju selekcji oraz wartości prawdopodobieństwa mutacji.

ZAD 2

W mieście, reprezentowanym za pomocą kwadratowej siatki, operujemy siecią pizzerii. Ich lokalizacje znajdują się w kwadratach o pozycjach (1,1), (2,6), (5,4), (7,1), albo jak pokazano na obrazku poniżej.



Zyski sieci zależą od budżetu wyłożonego na cztery pizzerie (B_1, B_2, B_3, B_4), gdzie każdy budżet jest w zakresie od **0 do 400** (nie może wyjść poza te wartości). Zysk jest liczony w następujący sposób:

$$Z = \sum_{i=0}^{8} \sum_{j=0}^{8} \frac{z_{i,j}^1 + z_{i,j}^2 + z_{i,j}^3 + z_{i,j}^4}{4} - (B_1 + B_2 + B_3 + B_4)^{1.15}$$

Gdzie $z_{i,j}^k$ jest dochodem k-tej pizzerii na polu (i, j). Innymi słowy, sumujemy **średnie zyski** z danego pola dla wszystkich pizzerii i odejmujemy od tego skorygowaną (potęgowaną) sumę budżetów — ze względu na konieczność rozbudowy lokali, większe trudności w utrzymaniu itp.

Zysk k-tej pizzerii na polu (i, j) liczymy natomiast następująco:

Najpierw liczymy d, czyli odległość taksówkarską (taxicab, Manhattan Distance) między pizzerią a tym polem.

- Jeżeli d jest **mniejsza** niż 2:

$$z = 1.3 \cdot \frac{B}{0.5d + 4}$$

- Jeżeli d jest **większa lub równa** 2:

$$z = \frac{B}{0.5d + 4}$$

Proszę zaprojektować **algorytm genetyczny**, który otrzyma jak najwyższą wartość funkcji $Z(B_1, B_2, B_3, B_4)$ i przedstawić, jak radzi sobie przy 10 wywołaniach (jak wygląda średnia wyniku i odchylenie standardowe).

Proszę samodzielnie wybrać **kodowanie** (rzeczywistoliczbowe, całkowitoliczbowe, binarne), wybrać **algorytm mutacji i selekcji**, oraz przeprowadzić **testy parametrów**.

In [143…

```
class PizzeriaSolution:
  shops = [(1,1), (2,6), (5,4), (7,1)]

  def __init__(self):
    self.B1 = random.randint(0,400)
    self.B2 = random.randint(0,400)
    self.B3 = random.randint(0,400)
    self.B4 = random.randint(0,400)
    self.genes = [self.B1, self.B2, self.B3, self.B4]

  @staticmethod
  def manhattan_distance(p1, p2):
    return abs(p1[0]-p2[0]) + abs(p1[1]-p2[1])

  def calculate_z(self):
    total_profit = 0.
    for i in range(8):
      for j in range(8):
        cell_profit = 0
        for k in range(4):
          d = self.manhattan_distance((i, j), PizzeriaSolution.shop
          if d<2:
            z = 1.3*self.genes[k]/(0.5*d+4)
          else:
            z = self.genes[k]/(0.5*d+4)
          cell_profit += z
        total_profit += cell_profit
    return total_profit/4

  def get_adaptation(self):
      profit = self.calculate_z()
      substract = sum(self.genes)**1.15
      return profit-substract

  def crossover(self, other_solution):
```

```
    crossober_point = random.randint(1,4)
    new_solution = PizzeriaSolution()
    new_solution.genes = self.genes[:crossober_point] + other_solut
    return new_solution

  # random resetting mutation: https://www.naukri.com/code360/libra
  def mutation(self, mutation_chance):
    for i in range(4):
      if random.random() < mutation_chance:
        self.genes[i] = random.randint(0,400)

  def __str__(self):
    return f'[{self.B1}, {self.B2}, {self.B3}, {self.B4}] : {self.g
```

In [144…

```
def run_algorithm(mutation_chance, selection_type,population_size ,
  population = [PizzeriaSolution() for i in range(population_size)]
  best_solution = PizzeriaSolution()
  best_solution_adaptation = 0.
  best_iteration_found = 0

  for iteration in range(iterations):
    adaptations = [p.get_adaptation() for p in population]
    local_best_solution = population[adaptations.index(max(adaptati
    if local_best_solution.get_adaptation() > best_solution_adaptat
      best_solution = local_best_solution
      best_solution_adaptation = local_best_solution.get_adaptation
      best_iteration_found = iteration

    if selection_type == 'roulette':
      roulette_wheel = adaptations
      for i in range(len(roulette_wheel)):
        roulette_wheel[i] -= min(adaptations)
        roulette_wheel[i] /= (max(adaptations)-min(adaptations))

      parents = [random.choices(population, weights=roulette_wheel,
      children = [p[0].crossover(p[1]) for p in parents]

    else:
      threshold_value = round(int(selection_type[-2:])/100*populati
      best_candidates = sorted(population, key=lambda x: x.get_adap
      parents = [random.choices(best_candidates, k=2) for i in rang
      children = [p[0].crossover(p[1]) for p in parents]

    for c in children:
        c.mutation(mutation_chance)
    population = children

  adaptations = [p.get_adaptation() for p in population]
  local_best_solution = population[adaptations.index(max(adaptation
  if local_best_solution.get_adaptation() > best_solution_adaptatio
    best_solution = local_best_solution
    best_solution_adaptation = local_best_solution.get_adaptation()

  if print_details:
    print(f'Best solution: {best_solution}')
```

```
  return best_solution, best_solution_adaptation
```

In [145…

```
import numpy as np

mutation_chances = [0.2, 0.5]
selection_types = ['roulette', 'treshold_30', 'treshold_60']
population_size = 100

for mutation_chance in mutation_chances:
  print(f'{print_dashes()} MUTATION CHANCE: {mutation_chance} {prin
  for selection_type in selection_types:
    print(f'{print_dashes()} SELECTION TYPE: {selection_type} {prin
    best_adaptations = []
    for i in range(10):
      print(f'RUN {i+1}', end = '\t')
      _, best_adaptation = run_algorithm(mutation_chance, selection
      best_adaptations.append(best_adaptation)
    avg_best_adaptation = np.mean(best_adaptations)
    std = np.std(best_adaptations)
    print(f'Średnia wyniku i odchylenie standardowe: {avg_best_adap
    print()
```

```
------------------------------ MUTATION CHANCE: 0.2 ----------------
--------------
------------------------------ SELECTION TYPE: roulette ------------
------------------
RUN 1   Best solution: [345, 71, 23, 102] : 121.09660317652231
RUN 2   Best solution: [112, 400, 172, 227] : 122.93776735901054
RUN 3   Best solution: [243, 236, 62, 344] : 121.16538203800212
RUN 4   Best solution: [246, 33, 214, 154] : 122.24953745673076
RUN 5   Best solution: [138, 196, 10, 29] : 120.51040947397803
RUN 6   Best solution: [359, 354, 138, 40] : 119.80783987617554
RUN 7   Best solution: [364, 253, 50, 370] : 119.30434222886674
RUN 8   Best solution: [217, 33, 14, 111] : 121.14245535747921
RUN 9   Best solution: [142, 219, 56, 236] : 123.19045273638312
RUN 10  Best solution: [330, 34, 361, 371] : 124.71441129397238
Średnia wyniku i odchylenie standardowe: 121.6119, 1.5735

------------------------------ SELECTION TYPE: treshold_30 ---------
--------------------
RUN 1   Best solution: [175, 330, 355, 69] : 126.35319995957025
RUN 2   Best solution: [107, 130, 335, 140] : 126.35528432627405
RUN 3   Best solution: [353, 59, 15, 49] : 126.2304593719839
RUN 4   Best solution: [173, 91, 10, 180] : 126.15126074123702
RUN 5   Best solution: [75, 47, 232, 169] : 126.35528432627405
RUN 6   Best solution: [118, 383, 356, 292] : 126.35483692511536
RUN 7   Best solution: [135, 390, 204, 241] : 125.73964197307691
RUN 8   Best solution: [241, 239, 317, 397] : 126.35483692511536
RUN 9   Best solution: [337, 80, 177, 366] : 126.35528432627405
RUN 10  Best solution: [393, 372, 121, 110] : 126.35483692511536
Średnia wyniku i odchylenie standardowe: 126.2605, 0.1861

------------------------------ SELECTION TYPE: treshold_60 ---------
--------------------
RUN 1   Best solution: [236, 262, 149, 350] : 123.81177236631811
RUN 2   Best solution: [74, 91, 39, 103] : 122.4066495696967
```

```
RUN 3   Best solution: [278, 271, 304, 143] : 121.10118117784543
RUN 4   Best solution: [173, 99, 192, 57] : 125.40544282360736
RUN 5   Best solution: [145, 189, 165, 69] : 123.83679133214639
RUN 6   Best solution: [145, 14, 129, 375] : 124.17098339595645
RUN 7   Best solution: [384, 30, 218, 188] : 124.248118227806
RUN 8   Best solution: [253, 271, 312, 96] : 121.92819561531257
RUN 9   Best solution: [63, 191, 172, 368] : 123.51921707068834
RUN 10  Best solution: [83, 30, 14, 316] : 125.31715775356088
Średnia wyniku i odchylenie standardowe: 123.5746, 1.3236

------------------------------ MUTATION CHANCE: 0.5 ----------------
--------------
------------------------------ SELECTION TYPE: roulette ------------
------------------
RUN 1   Best solution: [224, 331, 51, 91] : 117.01969742657673
RUN 2   Best solution: [143, 322, 79, 77] : 116.33238503303517
RUN 3   Best solution: [113, 73, 79, 53] : 121.00634801184515
RUN 4   Best solution: [96, 331, 160, 335] : 122.08731655924771
RUN 5   Best solution: [198, 189, 217, 354] : 119.28173174899246
RUN 6   Best solution: [360, 156, 0, 362] : 122.61060406451872
RUN 7   Best solution: [238, 165, 256, 380] : 116.19203146685118
RUN 8   Best solution: [105, 18, 22, 178] : 116.90520231886126
RUN 9   Best solution: [116, 282, 41, 249] : 118.05232059960645
RUN 10  Best solution: [178, 202, 294, 376] : 119.6780774454138
Średnia wyniku i odchylenie standardowe: 118.9166, 2.2643

------------------------------ SELECTION TYPE: treshold_30 ---------
---------------------
RUN 1   Best solution: [108, 310, 79, 23] : 120.82986531477741
RUN 2   Best solution: [263, 55, 178, 314] : 121.11652140034516
RUN 3   Best solution: [265, 246, 125, 394] : 124.55499314698659
RUN 4   Best solution: [169, 369, 32, 333] : 122.50366484836786
RUN 5   Best solution: [91, 183, 327, 63] : 123.57437565399823
RUN 6   Best solution: [343, 183, 342, 124] : 121.44505560661696
RUN 7   Best solution: [133, 62, 165, 349] : 122.72301670579259
RUN 8   Best solution: [349, 266, 157, 103] : 122.43884692022255
RUN 9   Best solution: [381, 341, 397, 322] : 123.19991616176299
RUN 10  Best solution: [113, 126, 68, 83] : 122.08037038253269
Średnia wyniku i odchylenie standardowe: 122.4467, 1.0902

------------------------------ SELECTION TYPE: treshold_60 ---------
---------------------
RUN 1   Best solution: [292, 147, 335, 63] : 117.10018394943677
RUN 2   Best solution: [283, 140, 247, 366] : 120.56788800936022
RUN 3   Best solution: [263, 121, 147, 174] : 124.74111198548258
RUN 4   Best solution: [387, 100, 76, 354] : 118.5696884187505
RUN 5   Best solution: [164, 144, 75, 124] : 117.33166049413194
RUN 6   Best solution: [39, 224, 186, 158] : 117.60914032477172
RUN 7   Best solution: [260, 89, 396, 61] : 119.0547096865671
RUN 8   Best solution: [73, 304, 208, 154] : 117.0796648884525
RUN 9   Best solution: [251, 62, 324, 282] : 119.16584516887599
RUN 10  Best solution: [28, 95, 126, 109] : 121.63695838601757
Średnia wyniku i odchylenie standardowe: 119.2857, 2.3192
```

Wnioski

Wybrałam kodowanie całkowitoliczbowe (chromosom o długości 4 w postaci kolejnych wartości budżetu dla kolejnych pizzerii), algorytm mutacji: random resetting mutation (bibliografia https://www.naukri.com/code360/library/mutation-in-genetic-algorithm). Wykonałam testy dla każdego z poznanych typów selekcji w poprzednim zadaniu oraz dla dwóch różnych wartości prawdopodobieństwa mutacji: 0.2 i 0.5. Przedstawiłam testy tych parametrów dla populacji o rozmiarze 100 i dla liczby iteracji 100. Największą wartość zwrócił algorytm threshold_30 przy prawdopodobieństwie mutacji 0.2. Dla tak dobranych parametrów przedstawiony został poniżej wynik optymalny oraz wartości optymalne budżetów pizzerii:

In [146…
```
best_solution, best_adaptation = run_algorithm(0.2, 'threshold_30',
```

```
Best solution: [162, 281, 67, 102] : 126.35454505261498
```

Dalsze zwiększanie liczby iteracji nie poprawie wyniku.