Kacper Dondziak

Temat: Algorytm genetyczny – labirynt (biblioteka deap)

1. Definicja problemu

1a. Struktura chromosomu

Chromosom został zdefiniowany jako 40 ciąg liczb z przedziału od 1 do 4 włącznie. Liczby odpowiadają kierunkom ruchu.

1 - lewo

2 - prawo

3 - Góra

4 - Dół

1b. Jak zakodowano problem

Mapa została przedstawiona jako 2 wymiarowa tablica liczb całkowitych. 1 oznacza ścianę, a 0 węzeł przez który można przejść.

```
map = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1],
    [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1],
    [1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
])
```

1c Który chromosom jest dobry?

Chromosom dobry to jest taki, który znajdzie ścieżkę prowadzącą od punktu początkowego do końcowego nie uderzając w ścianę.

1d. Maks. zakres funkcji Fitness

Funkcja fitness ma zakres od pewnej zależnej od inputu liczby ujemnej do 0, które oznacza, że algorytm znalazł ścieżkę oraz nie dostał punktów karnych przy kolizji ze ścianą. W przypadku funkcji fitness w wersji 2 maksymalnie dobre rozwiązanie będzie się znajdowało przy małej liczbie ujemnej (-5, 0).

1.e Mutacja, Krzyżowanie, elityzm

Do mutowania użyłem wbudowaną w bibliotekę funkcji mutUniformInt, która wymienia liczbę w chromosomie na inną z podanego przedziału.

Do krzyżowania użyłem wbudowaną w bibliotekę funkcji cxTwoPoint, która w solowym miejscu przecina chromosomy i wymienia przecięte części tworząc pokolenie potomne.

Do elityzmu użyłem wbudowaną w bibliotekę funkcje seTournament, która wybiera najlepsze osobniki, ale pozwala na utrzymanie różnorodności.

```
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", tools.mutUniformInt, indpb=mut_pb, low=1, up=4)
toolbox.register("select", tools.selTournament, tournsize=2)
```

2. Algorytmy

W projekcie zostały użyte 3 algorytmy:

- Genetyczny funkcja fitness wersja 1
- Genetyczny funkcja fitness wersja 2
- Przeszukiwanie wszerz

2.a Genetyczny funkcja fitness wersja 1

Algorytm bazuje na wyliczeniu wartości danego chromosomu wyliczając w jakim punkcie zakończy podróż przez labirynt, odejmując od wartości punkty karne za kolizje ze ścianą. Po wykryciu kolizji ze ścianą, ruch kolizyjny jest pomijany. Wartość chromosomu wyliczamy jako wartość bezwzględną z różnicy odległości punktu wyjścia od punktu zakończenia podróży.

```python
def get_distance_from_start(path):
    penalty_points = 0
    current_pos_x = 1
    current_pos_y = 1
    for i in range(len(path)):
        x = get_shifted_cord_x_start(current_pos_x, path[i])
        y = get_shifted_cord_y_start(current_pos_y, path[i])
        if map[current_pos_y][current_pos_x] == 1:
            penalty_points += 1
        else:
            current_pos_x, current_pos_y = x, y

    distance = get_absolute_distance_from_start(current_pos_x, current_pos_y)
    return get_weighted_value(distance, penalty_points)

def get_absolute_distance_from_start(x, y):
    return abs(10 - x) + abs(10 - y)

def get_weighted_value(distance, penalty):
    value = -(penalty * PENALTY_WEIGHT) - (distance * DISTANCE_WEIGHT)
    return value

def fitness_v2(individual):
    value = get_distance_from_start(individual) + get_distance_from_end(individual)
    return value,
```

2.b Genetyczny funkcja fitness wersja 2

Podobnie jak w wersji 1 funkcja fitness wylicza wartość chromosomu jako wartość bezwzględną między punktem zakończenia podróży i punktem końca. W wersji drugiej, karane jest wracanie do poprzednio odwiedzonego węzła. W związku z powyższym, funkcja fitness może przyjąć dla dobrego rozwiązania niewielką liczbę ujemną. Ta zmiana ma na celu zapobieganie przypadkom kręcenia się w miejscu.

```python
def get_penalty_for_moving_to_visited_node(path):
    penalty_points = 0
    for i in range(int(len(path) // 1.4)):
        if abs(path[i] - path[i + 1]) == 1:
            if not ((path[i] == 2 and path[i + 1] == 3) or (path[i] == 3 and path[i + 1] == 2)):
                penalty_points += 1
    return penalty_points


def fitness_v2(individual):
    value = get_distance_from_start(individual) - get_penalty_for_moving_to_visited_node(individual)
    return value,
```

2.c Przeszukiwanie wszerz

Algorytm oparłem na algorytmie znajdowania najkrótszej ścieżki metoda przeszukiwania wszerz. Polega on w skrócie na tych krokach:

- Przydziel wszystkim węzłą wartość identyfikacyjną zwna dalej id, liczbę wiekszą lub równą 0 tak aby była unikalna dla danej mapy i można było jednoznacznie z takiej liczby odnaleźć węzeł (np. dla mapy 12x12 x = 1, y = 2 -> (12 * y) + x = 24 + 1 = 25)
- Stwórz tablice o długości ilości węzłów wypełnioną wartością neutralną np. w tym przypadku -1, zwana dalej tablica odwiedzin.
- Dodaj do kolejki węzeł początkowy.
- Dopóki kolejka nie jest pusta lub nie znaleziono węzła końcowego:
    - Ściągnij węzeł z kolejki.
    - Rozpatrz wszystkie możliwe kroki od danego węzłą. Sprawdź czy kolejny węzeł nie był już odwiedzony. Jeżeli nie był to dodaj go do kolejki i oznacz jako odwiedzony. Do tablicy odwiedzin na pozycji id kolejnego węzła przypisz wartość id ściągniętego węzła.
- Jeżeli znaleziono węzeł końcowy to odczytywana jest ścieżka od węzła końcowego do węzła początkowego korzystając z tablicy odwiedzin.

```python
def add_nodes_to_queue(route_queue, map, route_map, x, y, map_width):
    if map[y][x + 1] != 1 and route_map[x + 1 + (y * map_width)] == 0:
        route_queue.put(x + 1 + (y * map_width))
        route_map[x + 1 + (y * map_width)] = x + (y * map_width)
    if map[y][x - 1] != 1 and route_map[x - 1 + (y * map_width)] == 0:
        route_queue.put(x - 1 + (y * map_width))
        route_map[x - 1 + (y * map_width)] = x + (y * map_width)
    if map[y + 1][x] != 1 and route_map[x + ((y + 1) * map_width)] == 0:
        route_queue.put(x + ((y + 1) * map_width))
        route_map[x + ((y + 1) * map_width)] = x + (y * map_width)
    if map[y - 1][x] != 1 and route_map[x + ((y - 1) * map_width)] == 0:
        route_queue.put(x + ((y - 1) * map_width))
        route_map[x + ((y - 1) * map_width)] = x + (y * map_width)


def search_route(map, start_pos_x, start_pos_y, end_pos_x, end_pos_y):
    print(map)
    map_width = len(map)
    route_map = [0 for i in range(map_width * map_width)]
    end_node_cords = end_pos_x + (map_width * end_pos_y)
    route_queue = queue.Queue()
    route_queue.put(start_pos_x + (start_pos_y * map_width))
    route_map[start_pos_x + (start_pos_y * map_width)] = -1
    while True:
        current_node = route_queue.get()
        print(f'{str(current_node)} : {current_node % map_width} : {current_node // map_width}')
        if current_node == end_node_cords:
            break
        add_nodes_to_queue(route_queue, map, route_map, current_node % map_width, current_node // map_width, map_width)
    print(route_map)
    read_route(map, start_pos_x, start_pos_y, end_pos_x, end_pos_y, route_map)


def read_route(map, start_pos_x, start_pos_y, end_pos_x, end_pos_y, path):
    map_width = len(map)
    result = []
    start_node = start_pos_x + (start_pos_y * map_width)
    current = end_pos_x + (end_pos_y * map_width)
    result.append(current)
    while start_node != current:
        current = path[current]
        result.append(current)
    result = result[::-1]
    get_directions_from_route(result, map)
```

3. Inputy

Wybrano 3 inputy. Dwa pierwsze od lewej sam wybrałem, a trzeci został wzięty ze skryptu laboratorium nr 2. Charakterystyczne dla mapy nr 1 jest to, że jest niemal pozbawiona przeszkód. Cechą drugiej jest to że istnieje tylko jedna poprawna ścieżka.

```
map = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
])
```

```
map = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1],
    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
])
```

```
map = np.array([
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
    [1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1],
    [1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 1],
    [1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
    [1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1],
    [1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1],
    [1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1],
    [1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1],
    [1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1],
    [1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 1],
    [1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1],
    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
])
```

Input nr 1 Input nr 2 Input nr 3

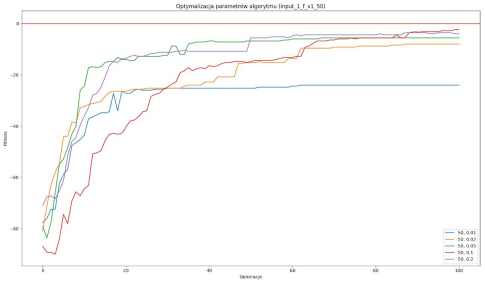
4. Optymalizacja algorytmów genetycznych

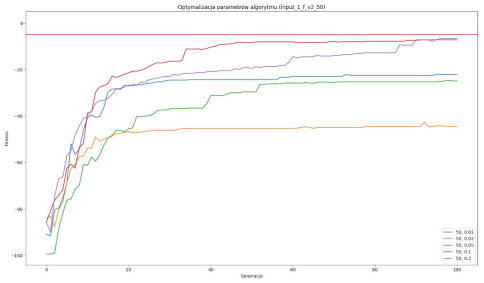
Celem tego rozdziału jest znalezienie optymalnych parametrów dla algorytmu genetycznego. Wnioski są wysuwane po 10 próbach. Dla uproszczenia optymalizacja jest prowadzona dla 100 generacji. Jednak algorytmy te nie są deterministyczne, więc wyniki mogą się różnić dla większej ilości prób. W przypadku funkcji fitness czerwona linia oznaczająca znalezione rozwiązanie znajduje się w punkcie -5, wynika to z tego, że rozwiązanie jest dobre ale występują “cofnięcia”. Z obserwacji działania algorytmu wyznaczono -5. Powyżej tej wartości optymalizowana jest ścieżka bez cofnięć.

4a. Input nr 1

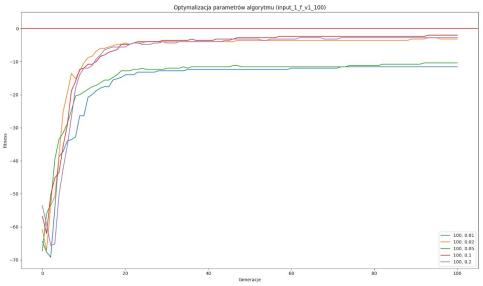
Fitness v1 **Fitness v2**

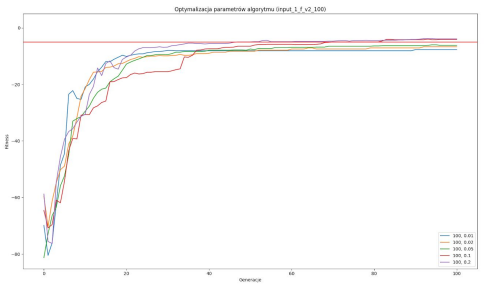
Populacja: 50



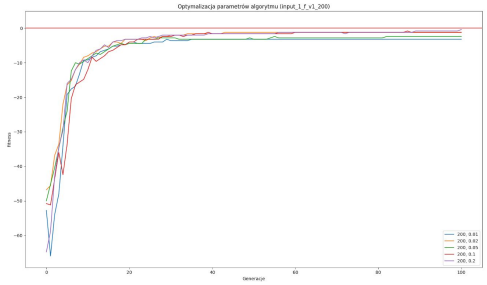


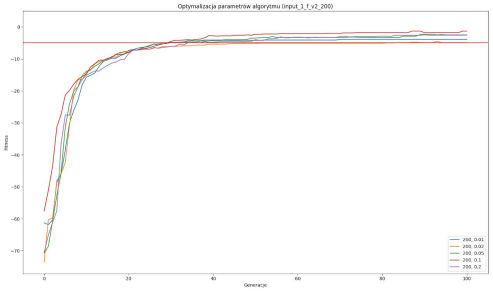
Populacja: 100



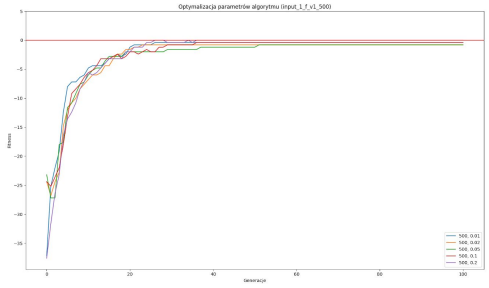


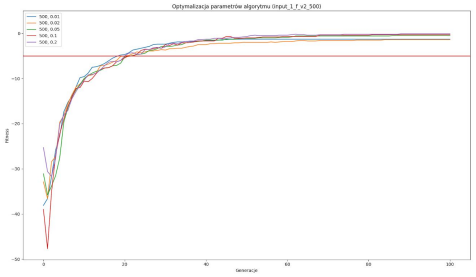
Populacja: 200





Populacja: 500



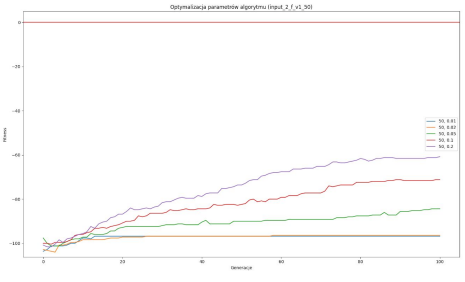


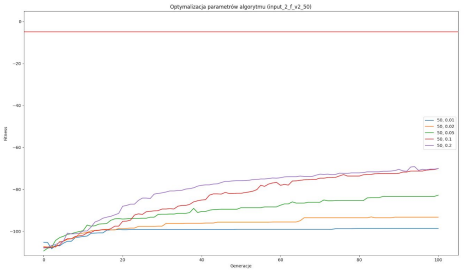
Dla tego inputu najbardziej optymalna wartość to populacja 200, 10% mutacji. Przy około 30 pokoleniach rozwiązanie jest osiągnięte.

4b. Input nr 2

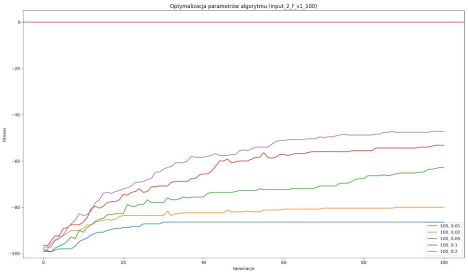
Fitness v1 **Fitness v2**

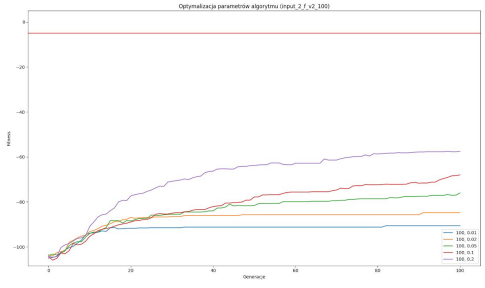
Populacja 50



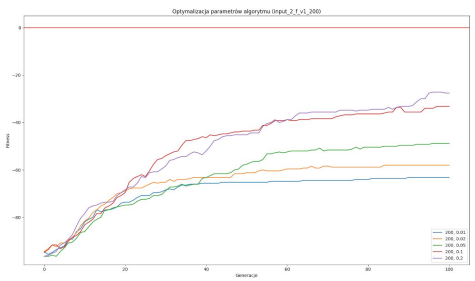


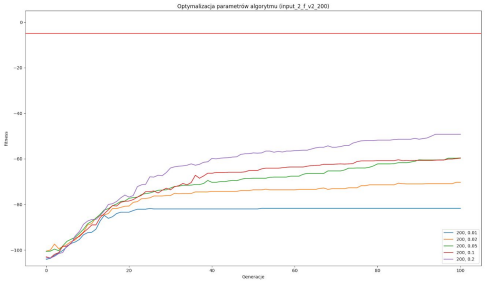
Populacja 100



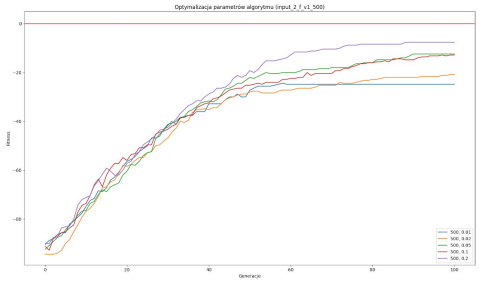


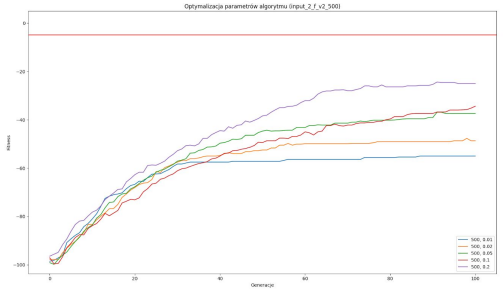
Populacja 200



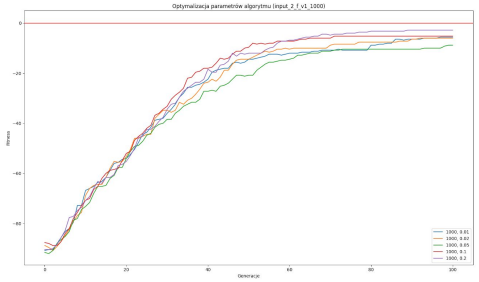


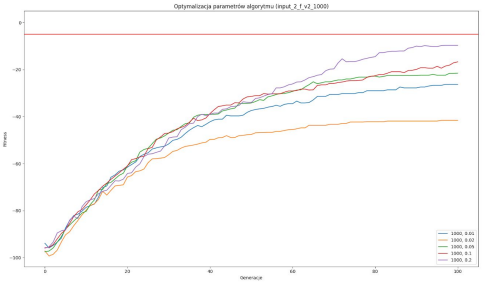
Populacja 500



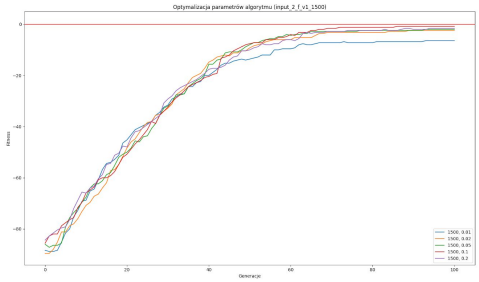


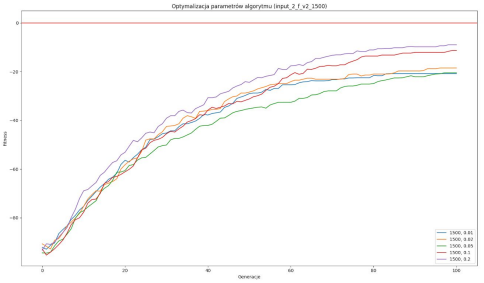
Populacja 1000





Populacja 1500



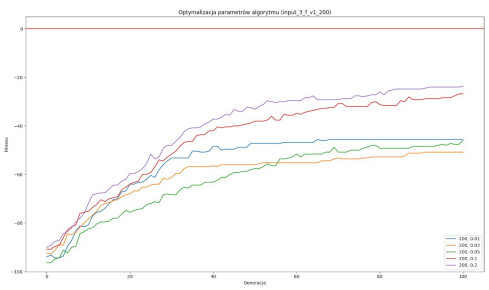


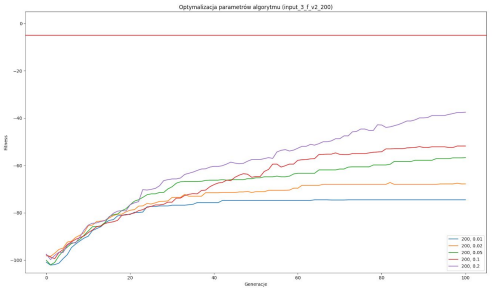
Dla tego inputu żaden algorytm nie dał rady.

4c. Input 3

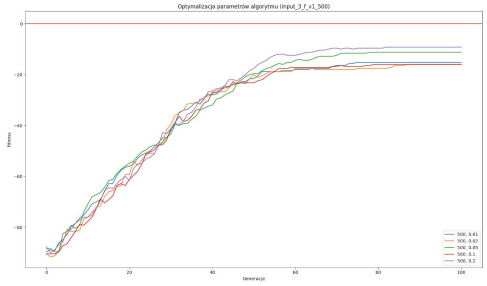
Fitness v1 **Fitness v**

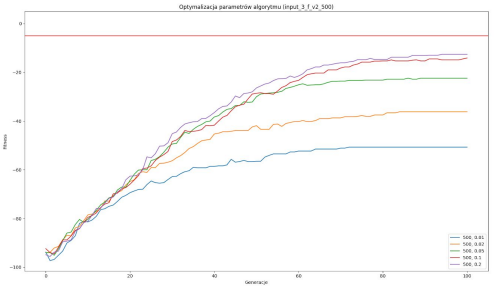
Populacja: 200



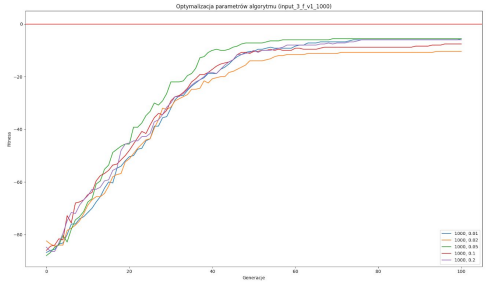


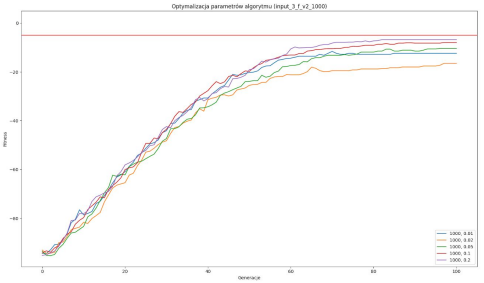
Populacja: 500



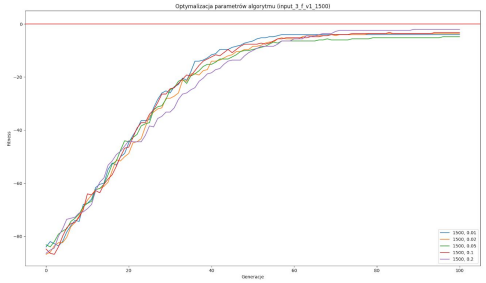


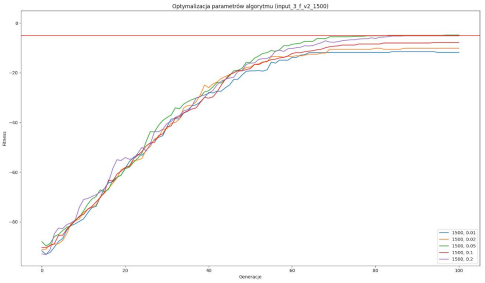
Populacja 1000





Populacja 1500





Rozwiązanie zostało znalezione dla wersji 2 przy 80 pokoleniach dla populacji 1500, mutacji 5%

5.Genetyczne vs Inne algorytmy

Pomiary czasu były prowadzone dla 20 prób dla optymalnych parametrów. Wynikiem jest średnia z 20 pomiarów.

	Input nr 1	Input nr 2	Input nr 3
Wersja 1	0.29812159538269045s (30 pokoleń, 200 populacja 0.1 mutacja)	brak	6.174892711639404 s (populacja 1500, mutacja, 5%, generacje 80)
Wersja 2	0.32920581102371216s (30 pokoleń, 200 populacja 0.1 mutacja)	brak	6.870333349704742 (populacja 1500, mutacja, 5%, generacje 80)
Przeszukiwanie wszerz	0.00044573545455932616 s	0.0001836061477661133 s	0.0003330826759338379 s

Algorytm genetyczny jest kilkadziesiąt razy wolniejszy (18538 razy dla inputu 3). Paradoksalnie przeszukiwanie wszerz było najważniejsze dla najłatwiejszego dla algorytmu genetycznego inputu.

6. Wnioski

Algorytmy genetyczne mogą być szybsze dla bardzo złożonych problemów, ale nie nadają się do rozwiązywania algorytmicznie prostych problemów.