Jan Mogielski 303899

Laboratorium 1
Zagadnienie przeszukiwania i podstawowe podejścia do niego

2 kwietnia 2023

Spis treści

1. Cel laboratorium

Celem laboratorium było napisanie kodu w języku Programowania Python, który znajduje najlepszego osobnika, który jest odpowiedzialny za sterowaniem dronem. Naszym celem jest osiągnięcie jak najwyższej wysokości bez rozbicia drona. Dron może mieść włączone lub wyłączone silniki przez maksymalnie 10 sekund, a potem poddaje się sile grawitacji i sile tarcia.

2. Implementacja

W pierwszej kolejności definiujemy klasę Solver(), dla której ustawiłem parametry podane w treści zadania. Funkcja get_parameters() zwraca wartości naszych parametrów, które możemy modyfikować w zależności od problemu.

```
'time_quantum': 0.1,
'on_time': 10.0,
'drone_acceleration': 30.0,
'gravity_acceleration': -10.0,
'drone_friction': -0.1,
'crash_velocity': 20.0,
'crash_penalty': -1500,
'population_size': 100,
'mutation_rate': 0.1,
'generations': 10
```

Listing 1: Działanie funkcji get_parameters()

Cały kod składa się z kilku głównych funkcji:

```
- def drone_flight(self, individual)
- def fitness_function(self, individual)
- def individual(self)
- def population(self)
- def new_individual(self, parent1, parent2)
- def mutate(self, individual)
- def roulette_wheel_selection(self, population, fitness_function_value)
- def reproduce(self, population)
- def Solve(self)
```

Listing 2: Funkcje wykorzystywane w programie

2.1. Funkcja drone_flight

Funkcja drone_flight symuluje lot naszego drona. Nasza funkcja przez pierwsze 10 sekund bierze osobnika, który ma długość 100 zer i jedynek, np: [0, 1, 1, 1, 0, 1, 0, 0.....], a następnie włącza i wyłącza silniki w dronie w zależności czy jest "1" lub "0". Po 10 sekundach silniki są wyłączane, a dron zaczyna opadać z powrotem na ziemię. Funkcja zwraca maksymalną wysokość uzyskaną przez naszego drona, prędkość w momencie powrotu na ziemię oraz czas lotu.

```
# simulation of drone flight
def drone_flight(self, individual):
    # starting values of the drone
    height = 0
    max_height = 0
    drone_velocity = 0
    flight_time = 0
    i = 0
    while height >= 0:  # for i in range(len(individual)):
        if flight_time >= self.on_time:
            drone_velocity += (self.gravity_acceleration - self.drone_friction * abs(
                drone_velocity)) * self.time_quantum
        elif i < 100:

            if individual[i] == 1:
                drone_velocity += (self.drone_acceleration + self.gravity_acceleration +
                ↪  self.drone_friction * abs(
                    drone_velocity)) * self.time_quantum
            else:
                drone_velocity += (self.gravity_acceleration - self.drone_friction *
                ↪  abs(
                    drone_velocity)) * self.time_quantum
            i += 1
        height += drone_velocity * self.time_quantum
        flight_time += 0.1
        max_height = max(height, max_height)
        print(max_height)
    return max_height, abs(drone_velocity), flight_time
```

Listing 3: Funkcja drone_flight()

2.2. Funkcja fitness_function

Funkcja fitness_function jest ściśle powiązana z drone_flight, ponieważ sprawda maksymalną wysokość uzyskaną przez drona i nakłada na nią karę, jeśli prędkość podczas lądowania jest za wysoka.

```
# fitness function for drone_flight
def fitness_function(self, individual):
    max_height, drone_velocity, flight_time = self.drone_flight(individual)
    if abs(drone_velocity) > self.crash_velocity:
        max_height += self.crash_penalty
    return max_height, abs(drone_velocity), flight_time
```

Listing 4: Funkcja fitness_function()

2.3. Funkcja individual

Funkcja individual zwraca nam osobnika o długości 100 składającego się z ”0” i ”1”, np: [1, 0, 0, 1, 1, 1, 0...].

```python
# create individual
def individual(self):
    return random.choices([0, 1], k=100)
```

Listing 5: Funkcja individual()

2.4. Funkcja population

Funkcja population tworzy populację osobników.

```python
# create population of individuals
def population(self):
    return [self.individual() for _ in range(self.population_size)]
```

Listing 6: Funkcja population()

2.5. Funkcja new_individual

Funkcja new_individual jest funkcją, która tworzy nowego osobnika z dwóch rodziców.

```python
# new individuals from old ones
def new_individual(self, parent1, parent2):
    where_to_divide = random.randint(1, 99)
    child1 = parent1[:where_to_divide] + parent2[where_to_divide:]
    # child2 = parent2[:where_to_divide] + parent1[where_to_divide:]
    return child1
```

Listing 7: Funkcja new_individual()

2.6. Funkcja mutate

Funkcja mutate jest funkcją, która poddaje mutacji danego osobnika zmieniając zero na jedynkę. Zmiana zachodzi w momencie, w którym random.random() jest mniejsze niż mutation_rate.

```python
# mutate individual by changing "0" and "1"
def mutate(self, individual):
    for i in range(len(individual)):
        if random.random() < self.mutation_rate:
            individual[i] = 1 - individual[i]
    return individual
```

Listing 8: Funkcja mutate

2.7. Funkcja roulette_wheel_selection

Funkcja roulette_wheel_selection odpowiada za...

2.8. Funkcja reproduce

Funkcja reproduce jest odpowiedzialna za stworzenie nowej populacji korzystając ze wcześniejszych funkcji. Każda populacja powinna być lepsza od poprzedniej.

```
# reproduce
def reproduce(self, population):
    new_population = []
    for i in range(self.population_size):
        parent1 = self.roulette_wheel_selection(population, self.fitness_function)
        parent2 = self.roulette_wheel_selection(population, self.fitness_function)
        child = self.new_individual(parent1, parent2)
        mutated_child = self.mutate(child)
        new_population.append(mutated_child)
    return new_population
```

Listing 9: Funkcja reproduce

2.9. Funkcja Solve

Funkcja Solve to funkcja łącząca wszystkie inne funkcje w celu znalezienia najlepszego osobnika ze wszystkich generacji. Iteruje ona przez wiele generacji.

```
def Solve(self):
    start_time = time.time()
    best_individual = None
    max_fitness_value = float('-inf')
    fitness_history = []
    population_solve = self.population()

    for i in range(self.generations):
        fitness_values = [self.fitness_function(individual)[0] for individual in
        ↪  population_solve]
        for j in range(len(population_solve)):
            if fitness_values[j] > max_fitness_value:
                max_fitness_value = fitness_values[j]
                best_individual = population_solve[j]
        population_solve = self.reproduce(population_solve)
        fitness_history.append(max_fitness_value)

    print(f"Best individual1:{best_individual}")
    print(f"Max fitness value1:{max_fitness_value}")
    print(f"Velocity for best individual:{self.fitness_function(best_individual)[1]}")
    print(f"Time flight for best
    ↪  individual:{self.fitness_function(best_individual)[2]}")
    print(f"Time elapsed:{time.time() - start_time:.2f}s")
    print("len populatiao_solve")
    print(len(fitness_history))

    plt.plot(range(self.generations), fitness_history)
    plt.xlabel("Generations")
    plt.ylabel("Best fitness")
    plt.show()
```

Listing 10: Funkcja Solve

3. Weryfikacja rozwiązania

Podczas tworzenia programu natknąłem się na wiele problemów, które musiałem rozwiązać. Podczas weryfikacji rozwiązania sprawdzałem działanie poszczególnych funkcji oraz typy danych.

3.1. Funkcja individual

W pierwszej kolejności sprawdziłem generowanie pojedynczego osobnika, w tym celu wywołuje funkcję individual(). W wyniku działania funkcji zwracany jest osobnik:

Individual: [1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1]
$< class' list' >$

Jak widzimy funkcja działa poprawnie, za każdym razem zwracany jest inny osobnik.

3.2. Funkcja population

Następnie sprawdzamy czy działa funkcja population:

Population: [[1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1], [1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1], ...]
$< class'list' >$

Jak widzimy funkcja działa poprawnie, za każdym razem zwracana jest cała populacja osobników.

3.3. Funkcja fitness_function

Sprawdzenie fitness_function wymaga podania jakiegoś individual, a funkcja powinna zwracać wysokość uzyskaną w danej próbie, prędkość końcową i cały czas lotu od wzbicia się w powietrze do upadku.

Fitness:(-834.8176361398729, 76.06381244996864, 33.80000000000021)

Pierwsza wartość to wysokość danej próby potrącona o karę, następna wartość to prędkość podczas lądowania, a ostatnia wartość to całkowity czas lotu. Całość działa poprawnie.

3.4. Funkcja new_individual

Do sprawdzenia tej funkcji jest wymagana populacja osobników, z której wybierzemy dwóch dowolnych rodziców.

New_Individual: [1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1]
$< class'list' >$

Nowy osobnik jest poprawnie generowany.

3.5. Funkcja mutate

Do mutacji osobnika potrzeba dowolnego osobnika, może się zdarzyć, że nie dochodzi do mutacji. Ale każdy osobnik ma długość 100, a mutation_rate o wartości 0.1, więc powinniśmy oczekiwać zmiany 10 zer na jedynki i odwrotnie (mniej więcej).

Individual: [0, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0]

Mutate: [0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0]

$< class'list' >$

Porównujące oryginalnego osobnika i zmutowanego, widzimy, że całość działa poprawnie.

3.6. Funkcja roulette_wheel_selection

3.7. Funkcja reproduce

Ostatnią funkcją do sprawdzenia była funkcja reproduce, która jest odpowiedzialna za reprodukowanie populacji. Do jej działania wykorzystujemy wszystkie inne funkcje.

Reproduce: [[1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1], [1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0], ...]

$< class'list' >$

Ta funkcja również działa poprawnie, zwraca nam ona nową populację z pożądanymi przez nas cechami.

3.8. Funkcja Solve

Jak widzimy pośrednie testy naszych funkcji wykazały ich poprawne działanie, dzięki temu możemy przejść do końcowego rozwiązania naszego problemu, czyli znalezienia idealnego osobnika, który osiągnie największa wysokość i postara się nie rozbić przy lądowaniu. W tym celu wykorzystamy funkcję Solve, w której będziemy tworzyć coraz lepsze generacje, czyli populacje składającą się z coraz lepiej dopasowanych osobników.

Dodatkowo w funkcji Solve liczymy czas jaki jest potrzebny na iteracje przez i generacji. Każdy osobnik w populacji jest poddawany ewaluacji jakości rozwiązania. Najbardziej interesującym parametrem jest wysokość i to ona definiuje, który osobnik jest tym najlepszym. Dla całej populacji w danej generacji szukamy najwyższego "dopasowania" (wysokości), jeśli znajdujemy najwyższą wysokość to ją zapisujemy i wiążemy z danym osobnikiem.

Dla najlepszego osobnika ze wszystkich generacji:
— na końcu wypisujemy naszego osobnika,
— osiągniętą przez niego wysokość,
— prędkość przy lądowaniu,
— czas lotu,
— czas potrzebny na sprawdzenie wszystkich generacji i ich reprodukcję,
— wykres przedstawiający postęp każdej następnej generacji

Wywołującą główną funkcję możemy stwierdzić czy rozwiązanie działa poprawnie.

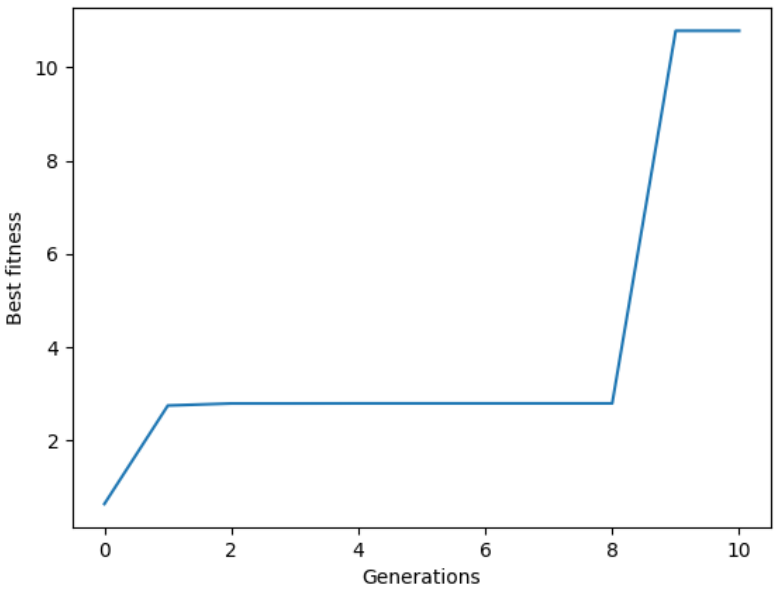
Best Individual: [1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1]

Max Fitness Value: 10.783644574153973

Velocity for Best Individual:7.850562229482857

Time flight for Best Individual:6.5999999999999925

Time Elapsed:106.06s



Rys. 1. Zależność wysokości od generacji

4. Wnioski

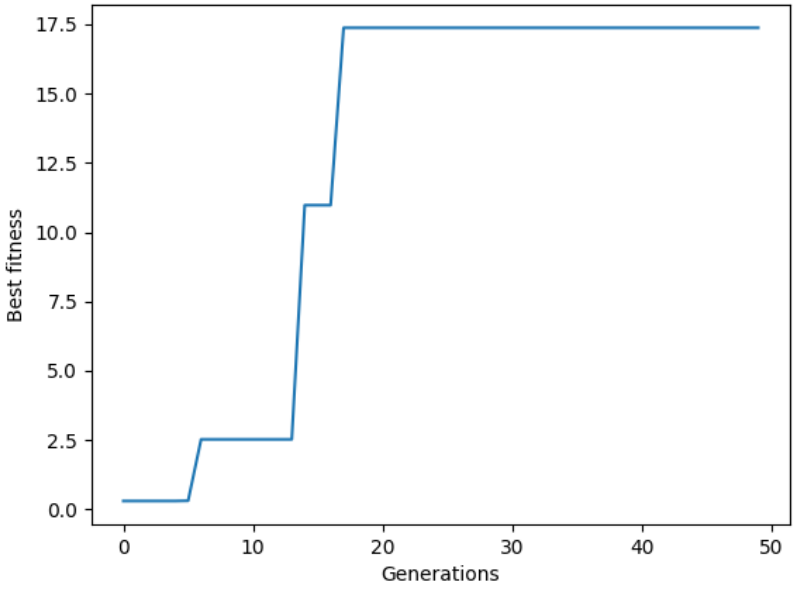
Wiedząc, że rozwiązanie działa możemy sprawdzić wpływ kilku parametrów na działanie algorytmu genetycznego. Najważniejsze parametry jakie możemy zmieniać to: wielkość populacji, współczynnik mutacji oraz ilość generacji.

W teorii im więcej osobników i generacji tym szybciej i skuteczniej powinniśmy znaleźć idealnego osobnika. Mutacja może być pomocna, ale może być też negatywna.

4.1. Parametr population_size

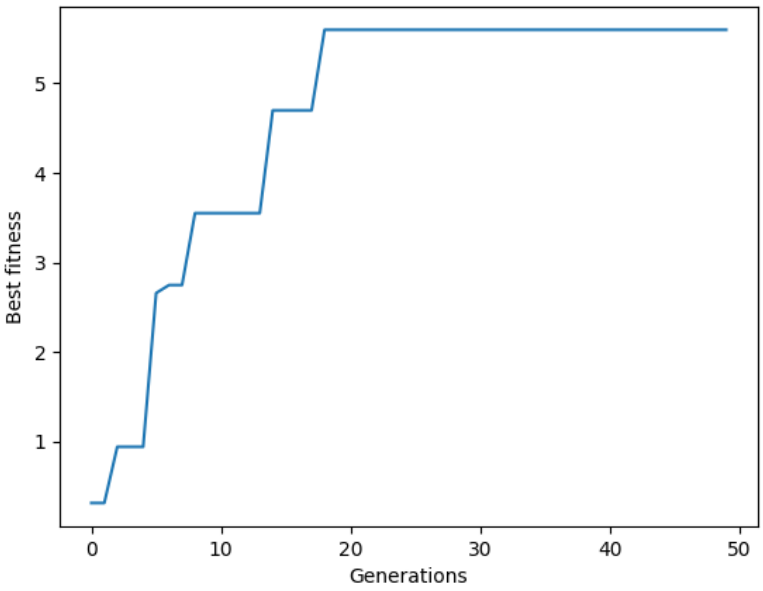
W pierwszej kolejności sprawdzę działanie algorytmu genetycznego w zależności od ilości osobników w populacji. Ilość generacji jest stała i wynosi 50.

4.1.1. population_size = 25



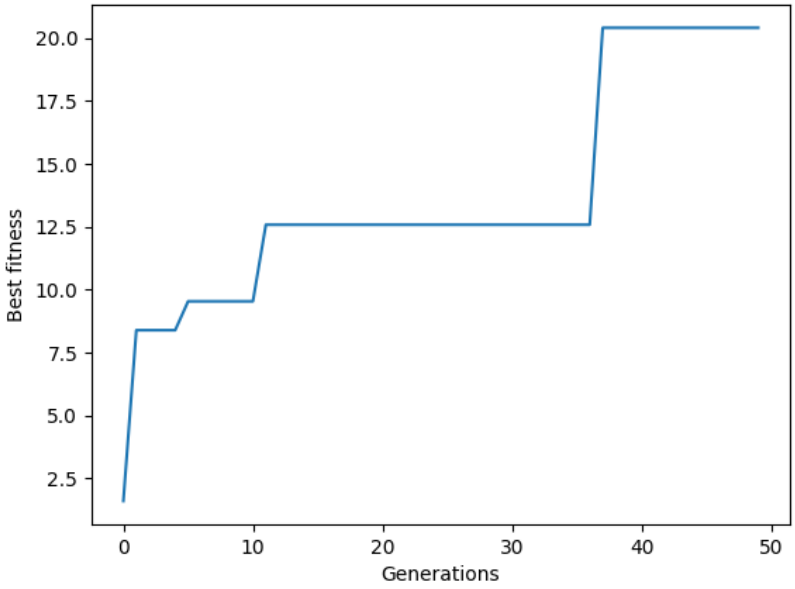
Rys. 2. Zależność wysokości od generacji dla populacji 25 osobników przy 50 generacjach

4.1.2. population_size = 50



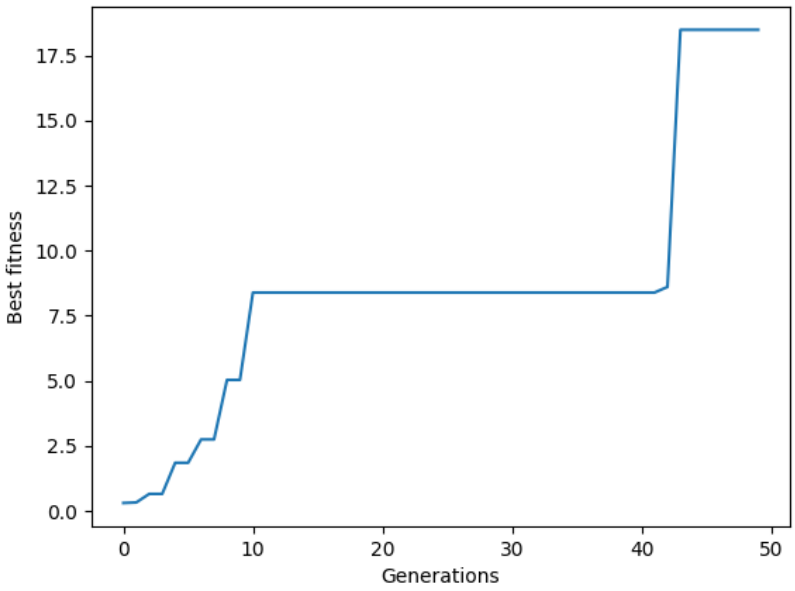
Rys. 3. Zależność wysokości od generacji dla populacji 50 osobników przy 50 generacjach

4.1.3. population_size = 75



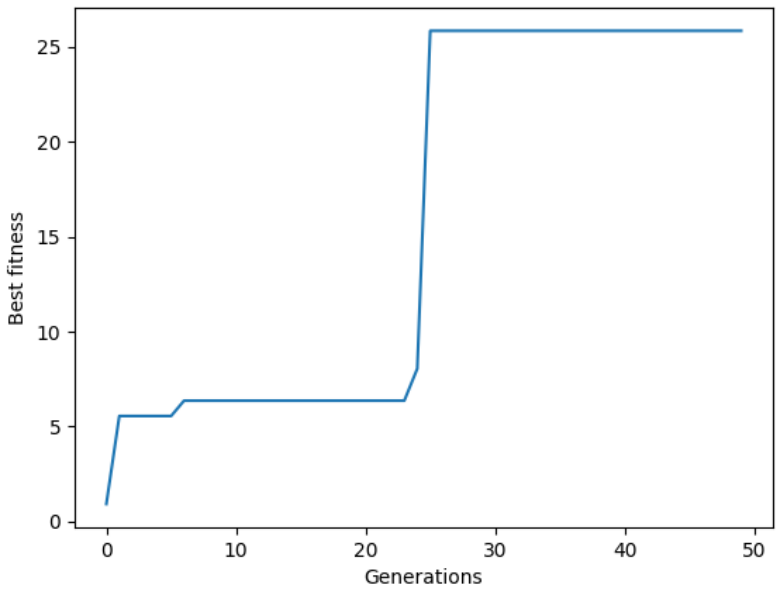
Rys. 4. Zależność wysokości od generacji dla populacji 75 osobników przy 50 generacjach

4.1.4. population_size = 100



Rys. 5. Zależność wysokości od generacji dla populacji 100 osobników przy 50 generacjach

4.1.5. population_size = 200



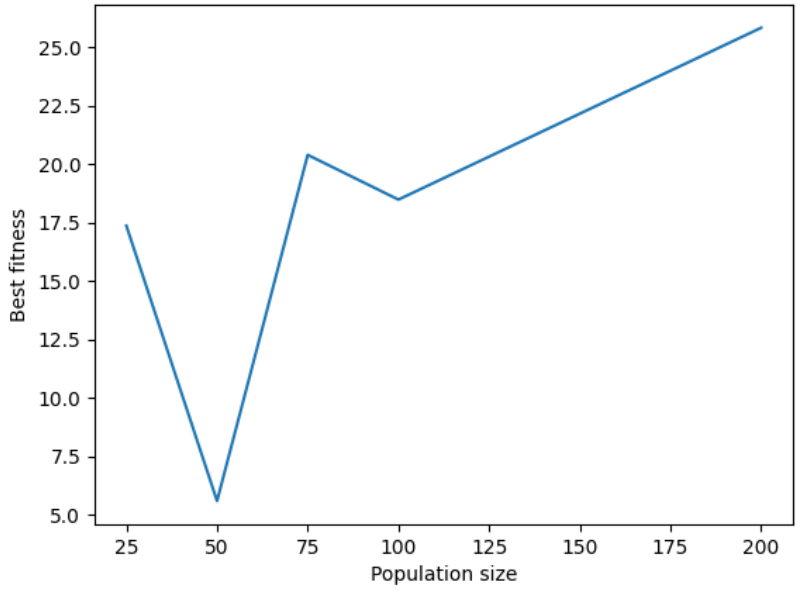
Rys. 6. Zależność wysokości od generacji dla populacji 200 osobników przy 50 generacjach

4.1.6. population_size = 400

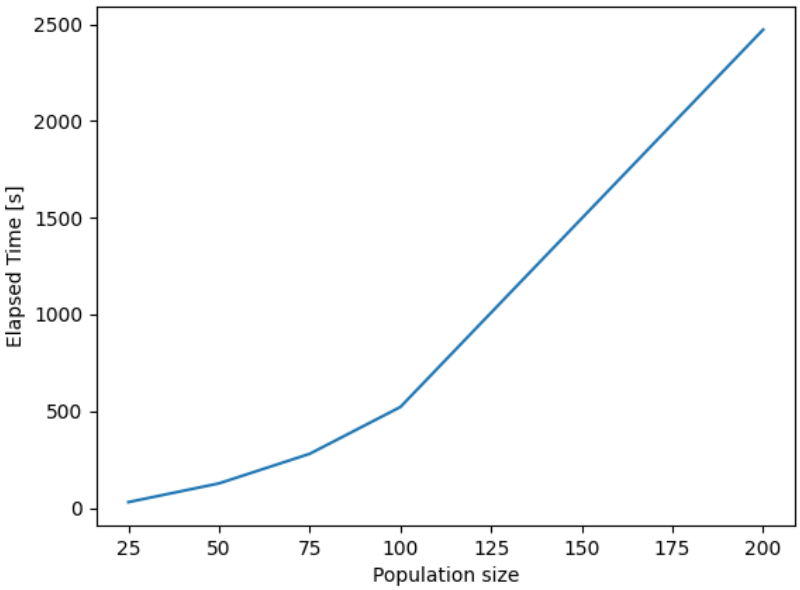
Dla tak dużej populacji działanie algorytmu jest trochę bezsensowne, ponieważ losując 400 osobników jest spora szansa, że znajdziemy idealnego osobnika na samym początku. Dodatkowo czas potrzebny na wykonanie całego algorytmu jest bardzo długi, ponieważ każdy z 400 osobników musi zostać sprawdzony i to musi zostać wykonane 50 razy, ponieważ tyle generacji ustawiliśmy. Tyle osobników pozwala na stworzenie elitarnej populacji, ale nie na tym polegało zadania.

4.1.7. Podsumowanie population_size

Ze względu na losowość tworzenia osobnika ciężko jest stwierdzić czy więcej osobników przyczynia się na znalezienie najlepszego osobnika. Dzięki większej ilości osobników możemy zmniejszyć ilość generacji potrzebnych na znalezienie najlepszego. Na pewno zwiększa się czas potrzebny na skończenie działania Algorytmu Genetycznego. Poniższy wykres jasno pokazuje, że nie zawsze większa liczba osobników jest lepsza, ale wraz ze wzrostem liczby osobników rośnie czas potrzebny na działanie algorytmu (wręcz eksponencjalnie).



Rys. 7. Zależność wysokości od ilości osobników

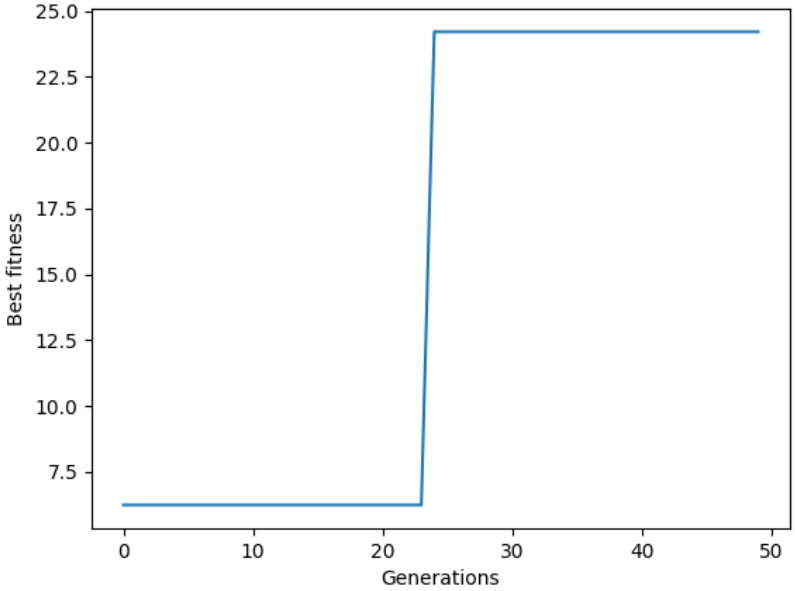


Rys. 8. Zależność czasu od ilości osobników

4.2. Parametr mutation_rate

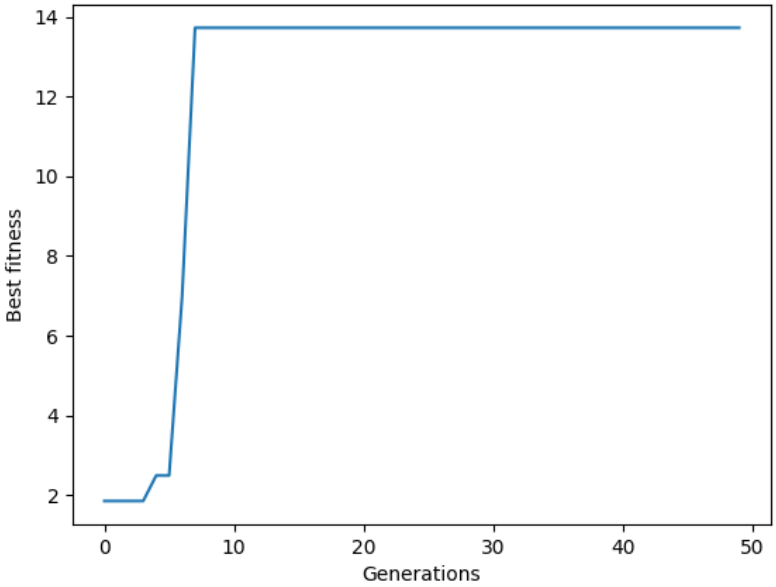
Dla sprawdzenie wpływu tego parametru przyjmujemy, że wielkość populacji to 100, a liczba generacji to 50. Większa mutacja powinna mocno wpływać na to jak wygląda lepsza populacja w każdej generacji, ponieważ bez mutacji większość osobników będzie wyglądać podobnie.

4.2.1. mutation_rate = 0.01



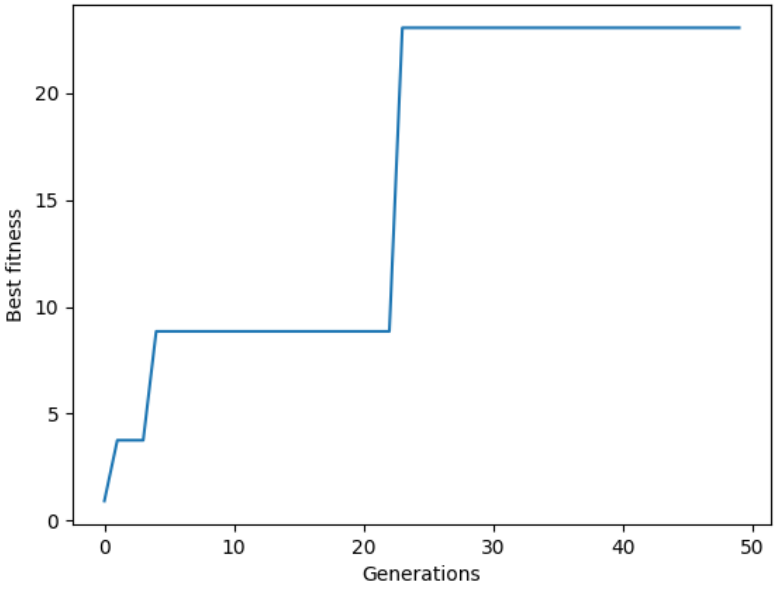
Rys. 9. Zależność wysokości od współczynnika mutacji przy 50 generacjach

4.2.2. mutation_rate = 0.1



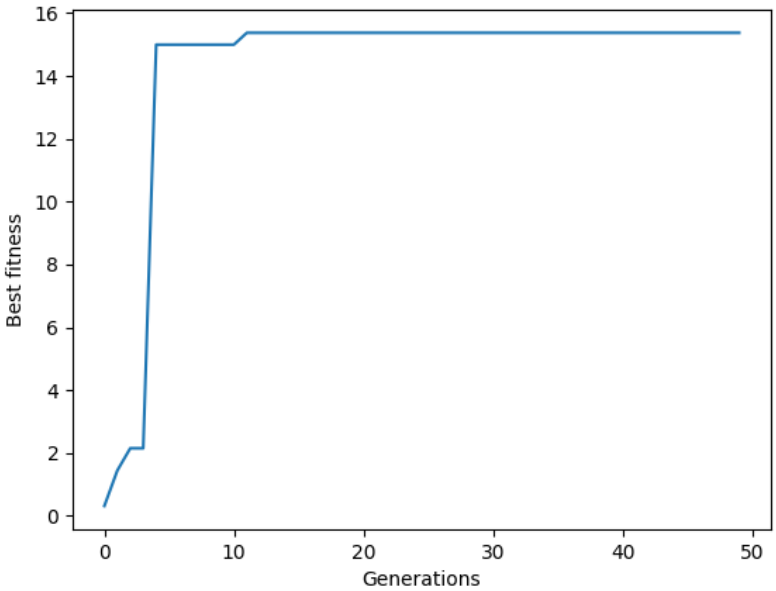
Rys. 10. Zależność wysokości od współczynnika mutacji przy 50 generacjach

4.2.3. mutation_rate = 0.2



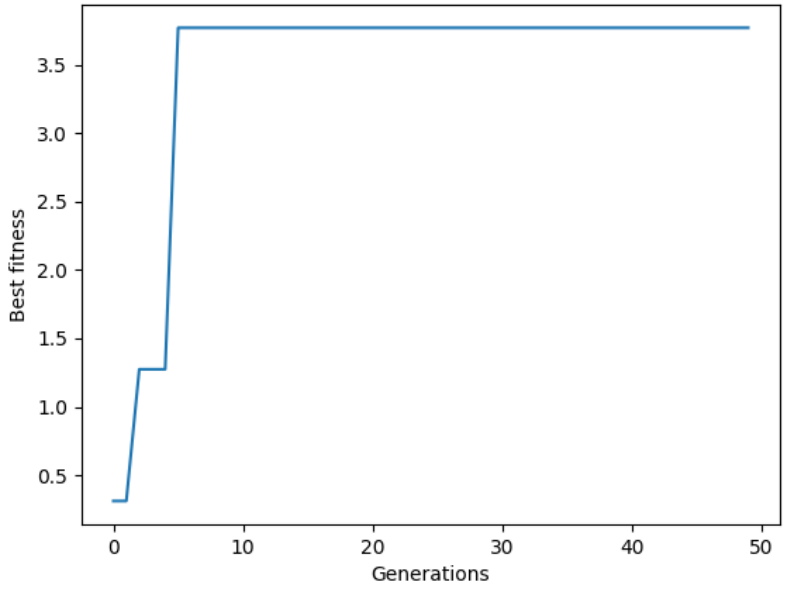
Rys. 11. Zależność wysokości od współczynnika mutacji przy 50 generacjach

4.2.4. mutation_rate = 0.4



Rys. 12. Zależność wysokości od współczynnika mutacji przy 50 generacjach

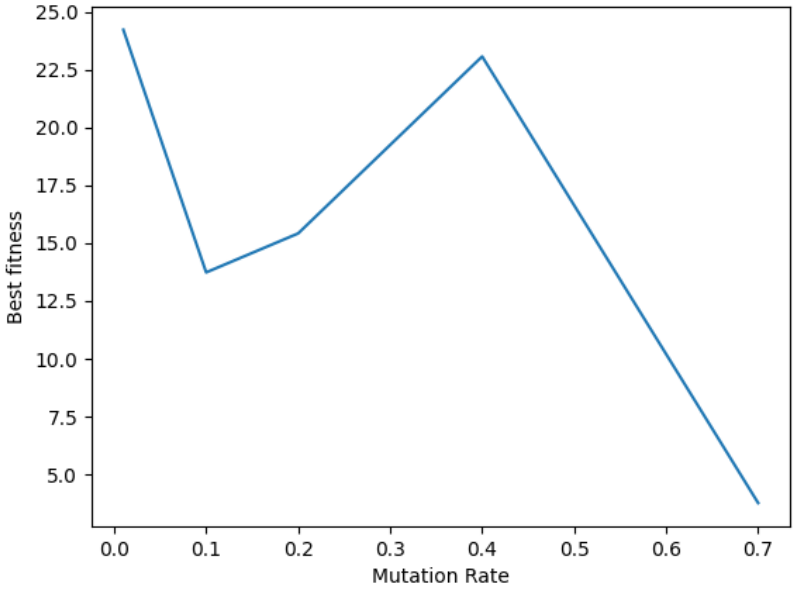
4.2.5. mutation_rate = 0.7



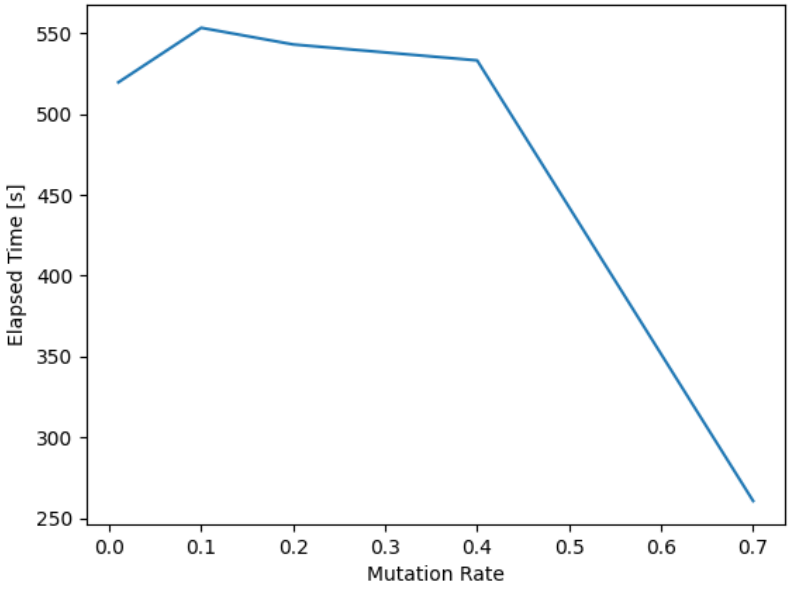
Rys. 13. Zależność wysokości od współczynnika mutacji przy 50 generacjach

4.2.6. Podsumowanie mutation_rate

Nadal problemem jest losowość tworzonych osobników, jednak możemy zaobserwować, że wysoka mutacja jest niepożądana. W przypadku wysokiej mutacji otrzymujemy osobniki, które są słabo przystosowane do naszego problemu, ale spada czas potrzebny na działanie algorytmu. Niska mutacja zapewnia nam niewielkie zmiany w wybranych osobnikach przez co cała populacja lepiej się dostosowuje do problemu. Duże zmiany w osobnikach zaburzają ten wzrost i przekazywanie dobrych genów.



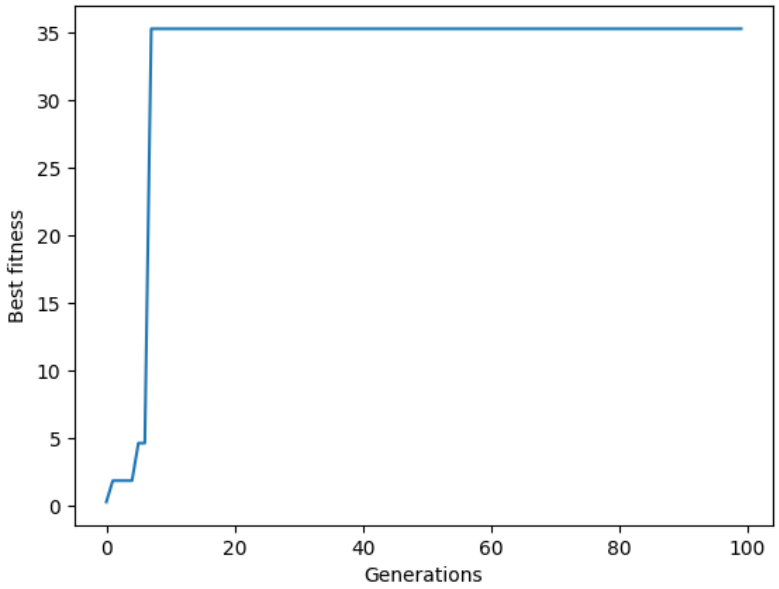
Rys. 14. Zależność wysokości od współczynnika mutacji



Rys. 15. Zależność czasu od współczynnika mutacji

4.3. Parametr generations

Ostatnim istotnym parametrem jest ilość generacji, które wpływają na reprodukcję całej populacji im więcej populacji tym lepiej dopasowane będą nasze osobniki. Im więcej osobników w populacji tym mniej generacji potrzebujemy do znalezienia tego najlepszego. Dodatkowo w pewnym momencie napotykamy problem, gdzie dalsza reprodukcja nie przynosi większych zmian dla naszego problemu. Dla 100 generacji, w której jest 100 osobników stosunkowo szybko obserwujemy osiągnięcie maksymlanej wysokości.



Rys. 16. Zależność wysokości od ilości generacji

Usprawnienia jakie widzę przy takich testach to zawsze zaczynanie od jednego osobnika, który osiąga jakąś wysokość. Z takiego punktu startowego byłoby łatwiej określić skuteczność zmian w określonych parametrach. Dodatkowo kara mogłaby być bardziej dotkliwa za przekroczenie prędkości co skutkowałoby ostrożniejszą selekcją osobników. Działanie takiego algorytmu dla 100 generacji po 100 osobników w populacji to czas 1080.97s, czyli 18 minut. Im większe te współczynniki tym więcej czasu zajmie działanie.