# Podstawy sztucznej inteligencji – Sprawozdanie

## Autor: Dawid Dąbrowski

## Rozwiązanie problemu plecakowego za pomocą algorytmu genetycznego

---

**1. Wprowadzenie**

Problem plecakowy można sobie wyobrazić jako dobrze znaną sytuację pakowania się w długą podróż. Wybierając się w taką podróż mamy wiele rzeczy, które chcielibyśmy zabrać, ale ogranicza nas pojemność walizki. Każdy przedmiot ma dla nas określoną wartość, jaką wnosi do podróży, a jednocześnie posiada rozmiar i wagę, co powoduje, że musi rywalizować o miejsce z innymi rzeczami. Jest to idealny przykład problemu plecakowego w codziennym życiu, który jest uważany za jeden z najstarszych i najlepiej zbadanych problemów wyszukiwania kombinatorycznego.

Formalnie rzecz ujmując, problem plecakowy składa się z elementów:

- Zbiór przedmiotów, z których każdy ma przypisaną określoną wartość oraz określoną wagę.
- Torba/plecak/pojemnik (tzw. „plecak") o określonej maksymalnej pojemności wagowej.

Naszym celem jest wybranie takiego zestawu przedmiotów, który zapewni maksymalną łączną wartość, nie przekraczając przy tym całkowitej pojemności wagowej plecaka.

---

**2. Opis teoretyczny algorytmu genetycznego**

Algorytm genetyczny opiera się na modelowaniu procesów naturalnej selekcji i ewolucji. Poniżej przedstawiono podstawowe elementy algorytmu:

**2.1 Reprezentacja rozwiązań**

Rozwiązania problemu są kodowane jako chromosomy. W przypadku problemu plecakowego każdy chromosom jest tablicą binarną (0 i 1), gdzie:

- 1 oznacza, że dany przedmiot został wybrany do plecaka.
- 0 oznacza, że dany przedmiot nie został wybrany.

**3.2 Populacja początkowa**

Na początku generowana jest losowa populacja początkowa o ustalonej liczbie osobników.

**3.3 Funkcja dopasowania (fitness function)**

Funkcja dopasowania oblicza jakość danego rozwiązania. Wartość funkcji fitness jest sumą wartości przedmiotów wybranych do plecaka, pod warunkiem, że łączna waga nie przekracza ustalonego limitu. Rozwiązania naruszające ograniczenie wagi otrzymują bardzo niską wartość fitness.

### 3.4 Operatory genetyczne

1. **Selekcja**: Wyboru rodziców do krzyżowania dokonuje się na podstawie ich wartości fitness.
2. **Krzyżowanie (crossover)**: Dwa chromosomy rodziców są łączone w celu stworzenia potomków.
3. **Mutacja**: Każdy gen chromosomu może ulec losowej zmianie (0 na 1 lub 1 na 0) z określonym prawdopodobieństwem.

### 3.5 Kryteria zakończenia

Algorytm kończy działanie po osiągnięciu maksymalnej liczby iteracji lub braku poprawy najlepszego rozwiązania przez określoną liczbę generacji.

---

## 4. Opis implementacji programu

### 4.1 Język i środowisko

Do implementacji wykorzystano język Python.

### 4.2 Struktura kodu

Program składa się z następujących modułów:

- **Generowanie populacji**: Funkcja inicjalizująca losową populację.

```python
def create_initial_population(self) -> List[List[int]]:
    """Create random initial population of binary chromosomes."""
    return [[random.randint(0, 1) for _ in self.items]
            for _ in range(self.population_size)]
```

- **Funkcja dopasowania**: Oblicza wartość fitness dla każdego osobnika. Reprezentowana jako suma wartości przedmiotów wybranych do plecaka.

```python
def calculate_fitness(self, chromosome: List[int]) -> float:
    """Calculate fitness value for a chromosome."""
    total_weight = sum(c * item.weight for c, item in zip(chromosome, self.items))
    if total_weight > self.capacity:
        return 0.0
    return sum(c * item.value for c, item in zip(chromosome, self.items))
```

- **Operatory genetyczne**: Implementacja selekcji metodą turniejową, krzyżowania jednopunktowego oraz losowej mutacji genów przy odpowiednim prawdopodobieństwie mutacji.

```python
def tournament_selection(self, population: List[List[int]]) -> List[int]:
    """Select chromosome using tournament selection."""
    tournament = random.sample(population, self.tournament_size)
    return max(tournament, key=self.calculate_fitness)

def crossover(self, parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Perform single-point crossover between parents."""
    point = random.randint(1, len(self.items)-1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(self, chromosome: List[int]) -> List[int]:
    """Apply mutation to chromosome."""
    return [1 - gene if random.random() < self.mutation_rate else gene
            for gene in chromosome]
```

- **Główna pętla algorytmu**: Obsługuje proces ewolucji i zapisuje wyniki.

  **Schemat działania**:

  1. Inicjacja – losowa generacja populacji początkowej

  2. Tworzenie kolejnych pokoleń dopóki nie zostanie osiągnięty warunek stopu (z góry zdefiniowana liczba pokoleń):

     a) Obliczenie funkcji przystosowania.
     b) Przeprowadzenie selekcji.
     c) Zastosowanie operatorów genetycznych.

```python
def solve(self) -> Tuple[List[int], float]:
    """
    Solve the knapsack problem using genetic algorithm.

    Returns:
        Tuple containing best solution and its fitness value
    """
    population = self.create_initial_population()
    best_solution = None
    best_fitness = 0

    for generation in range(self.generations):
        new_population = []
        current_best = max(population, key=self.calculate_fitness)
        current_best_fitness = self.calculate_fitness(current_best)

        if current_best_fitness > best_fitness:
            best_solution = current_best
            best_fitness = current_best_fitness

        self.best_fitness_history.append(best_fitness)
        new_population.append(current_best)

        while len(new_population) < self.population_size:
            parent1 = self.tournament_selection(population)
            parent2 = self.tournament_selection(population)
            child1, child2 = self.crossover(parent1, parent2)
            new_population.extend([self.mutate(child1), self.mutate(child2)])

        population = new_population[:self.population_size]
        logging.info(f"Generation {generation}: Best fitness = {best_fitness}")

    return best_solution, best_fitness
```

### 4.3 Parametry algorytmu

Parametry algorytmu są przyjmowane w konstruktorze klasy implementującej rozwiązanie algorytmu.

```python
class GeneticKnapsack:
    """Genetic algorithm implementation for solving the knapsack problem."""

    def __init__(self, items: List[Item], capacity: float,
                 population_size: int = 100, generations: int = 10000,
                 mutation_rate: float = 0.1, tournament_size: int = 3) -> None:
        """
```

### 4.4 Obsługa danych wejściowych

Dane wejściowe są wprowadzane jako lista przedmiotów. Przedmiot jest reprezentowany jako obiekt klasy Item, posiadający pola wagi i wartości.

```python
@dataclass
class Item:
    """Represents an item in the knapsack problem."""
    weight: float
    value: float
```

---

## 5. Wyniki

### 5.1 Wyniki działania programu

Przykład wyniku dla powyższego zestawu danych:

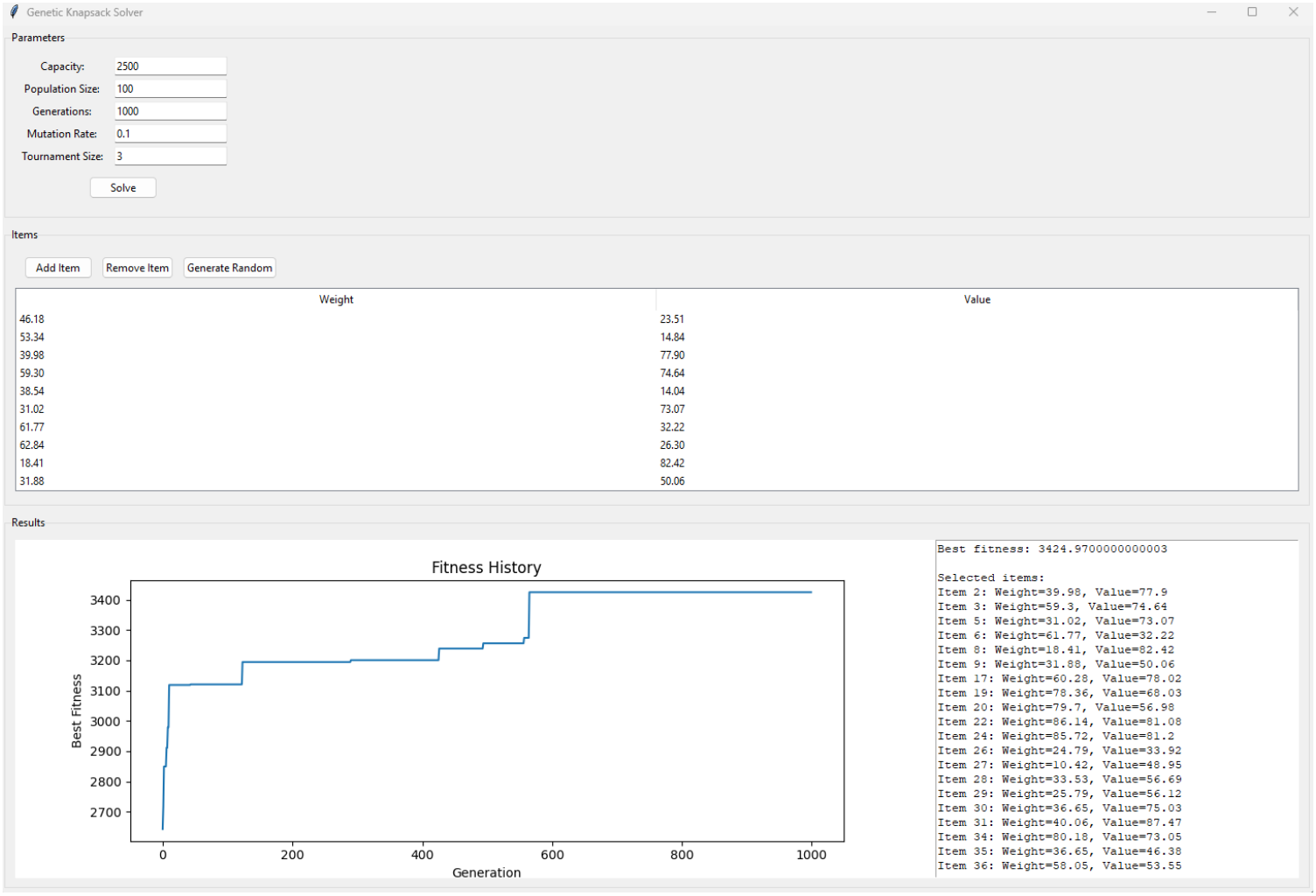
- Objętość plecaka: 2500
- Objętość i waga przedmiotów z zakresu 10 do 90
- Ilość przedmiotów: 100
- Wielkość turnieju: 2
- Prawdopodobieństwo mutacji: 0.1

```
INFO:root:Generation 988: Best fitness = 3582.9639031280567
INFO:root:Generation 989: Best fitness = 3582.9639031280567
INFO:root:Generation 990: Best fitness = 3582.9639031280567
INFO:root:Generation 991: Best fitness = 3582.9639031280567
INFO:root:Generation 992: Best fitness = 3582.9639031280567
INFO:root:Generation 993: Best fitness = 3582.9639031280567
INFO:root:Generation 994: Best fitness = 3582.9639031280567
INFO:root:Generation 995: Best fitness = 3582.9639031280567
INFO:root:Generation 996: Best fitness = 3582.9639031280567
INFO:root:Generation 997: Best fitness = 3582.9639031280567
INFO:root:Generation 998: Best fitness = 3582.9639031280567
INFO:root:Generation 999: Best fitness = 3582.9639031280567
Best solution: [1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1,
1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 1, 1
, 0, 1, 1, 0, 1, 0]
Total value: 3582.9639031280567
Selected items:
Item 0: Weight=75.70779796620118, Value=80.18383699987037
Item 4: Weight=32.29664001123347, Value=85.66638717578189
Item 5: Weight=26.307345280717804, Value=56.21068487471747
Item 7: Weight=83.54986807532855, Value=84.06316709019595
Item 8: Weight=28.62585447218203, Value=83.66715936689641
Item 10: Weight=16.459831963121072, Value=62.261241962198504
Item 11: Weight=73.06707290355354, Value=69.23215548991729
Item 14: Weight=37.81713292021674, Value=59.02709227582284
Item 15: Weight=26.495206166203637, Value=23.622578266087764
Item 18: Weight=10.595476533746808, Value=41.73952275214209
Item 19: Weight=15.273467076930558, Value=54.11851801058447
Item 20: Weight=25.990444040523162, Value=58.160380053605834
Item 21: Weight=87.11796800629156, Value=78.41255408828185
```

*Rysunek 1 Wycinek działania programu w konsoli*

### 5.2 Wizualizacja wyników

Do wizualizacji wyników przygotowałem również prosty interfejs użytkownika:



# Rozwiązanie problemu zarządzania przydziałem zadań w systemie wieloprocesorowym

---

### 1. Wprowadzenie

Problem zarządzania przydziałem zadań w systemie wieloprocesorowym, znany również jako Load Balancer, polega na przypisaniu zadań do dostępnych procesorów w sposób, który minimalizuje czas realizacji najdłuższego zadania, zwany makespan. Celem jest równomierne rozłożenie obciążenia pomiędzy procesory, aby zminimalizować czas oczekiwania na zakończenie obliczeń.

---

### 2. Opis teoretyczny algorytmu genetycznego

Algorytm genetyczny w tym przypadku będzie analogiczny do problemu plecakowego.

#### 2.1 Reprezentacja rozwiązań

Rozwiązania problemu są kodowane jako chromosomy, które stanowią reprezentację przydziału zadań do procesorów. Każdy chromosom to tablica, w której elementy wskazują, do którego procesora jest przypisane zadanie. Wartość każdego elementu tablicy (0..N) wskazuje numer procesora, do którego zostało przypisane zadanie np:

- **0, 1, 2, ...** oznaczają numer procesora przypisanego do danego zadania (indeksy procesorów zaczynają się od 0).

#### 3.2 Populacja początkowa

Na początku generowana jest losowa populacja początkowa o ustalonej liczbie osobników.

#### 3.3 Funkcja dopasowania (fitness function)

Funkcja dopasowania ocenia jakość danego rozwiązania, obliczając makespan dla przydziału zadań. Makespan to maksymalny czas pracy spośród wszystkich procesorów, który jest obliczany na podstawie sumy czasów wykonania zadań przypisanych do danego procesora, uwzględniając wydajność (mnożnik obciążenia) każdego procesora.

Funkcja fitness w przypadku Load Balancera minimalizuje wartość makespan, co oznacza, że niższa wartość fitness wskazuje na lepsze rozwiązanie. Im bardziej równomiernie zadania są rozłożone między procesory, tym lepsze jest rozwiązanie.

#### 3.4 Operatory genetyczne

4. **Selekcja**: Wyboru rodziców do krzyżowania dokonuje się na podstawie ich wartości fitness.
5. **Krzyżowanie (crossover)**: Dwa chromosomy rodziców są łączone w celu stworzenia potomków.
6. **Mutacja**: Każdy gen chromosomu może ulec losowej zmianie przydziału procesora z określonym prawdopodobieństwem.

#### 3.5 Kryteria zakończenia

Algorytm kończy działanie po osiągnięciu maksymalnej liczby iteracji lub braku poprawy najlepszego rozwiązania przez określoną liczbę generacji.

## 4. Opis implementacji programu

### 4.1 Język i środowisko

Do implementacji wykorzystano język Python. Część kodu została użyta ponownie z problemu plecakowego.

### 4.2 Struktura kodu

Program składa się z następujących modułów:

- **Generowanie populacji**: Funkcja inicjalizująca losową populację.

```python
def create_initial_population(self) -> List[List[int]]:
    """Create random initial population of chromosomes."""
    return [[random.randint(0, len(self.processors) - 1) for _ in self.tasks]
            for _ in range(self.population_size)]
```

- **Funkcja dopasowania**: Oblicza wartość dopasowania dla każdego osobnika. Reprezentowana jako największy czas wykonania zadań przez procesory. Niższa wartość wskazuje na lepsze rozwiązanie

```python
def calculate_fitness(self, chromosome: List[int]) -> float:
    """..."""
    processor_times = [0.0] * len(self.processors)

    for task_idx, processor_idx in enumerate(chromosome):
        task_time = self.tasks[task_idx].execution_time
        processor_multiplier = self.processors[processor_idx].multiplayer
        processor_times[processor_idx] += task_time * processor_multiplier

    return max(processor_times)
```

- **Operatory genetyczne**: Implementacja selekcji metodą turniejową, krzyżowania jednopunktowego oraz losowej mutacji genów przy odpowiednim prawdopodobieństwie mutacji.

```python
def tournament_selection(self, population: List[List[int]]) -> List[int]:
    """Select chromosome using tournament selection."""
    tournament = random.sample(population, self.tournament_size)
    return min(tournament, key=self.calculate_fitness)

def crossover(self, parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Perform single-point crossover between parents."""
    point = random.randint(1, len(self.tasks)-1)
    child1 = parent1[:point] + parent2[point:]
    child2 = parent2[:point] + parent1[point:]
    return child1, child2

def mutate(self, chromosome: List[int]) -> List[int]:
    """Apply mutation to all tasks with probability."""
    return [random.randint(0, len(self.processors) - 1) if random.random() < self.mutation_rate else gene
            for gene in chromosome]
```

- **Główna pętla algorytmu**: Obsługuje proces ewolucji i zapisuje wyniki.

  **Schemat działania**:

  1. Inicjacja – losowa generacja populacji początkowej

  2. Tworzenie kolejnych pokoleń dopóki nie zostanie osiągnięty warunek stopu (z góry zdefiniowana liczba pokoleń):

     d) Obliczenie funkcji przystosowania.
     e) Przeprowadzenie selekcji.
     f) Zastosowanie operatorów genetycznych.

```python
def solve(self) -> Tuple[List[int], float]:
    """..."""
    population = self.create_initial_population()
    best_solution = None
    best_fitness = float('inf')

    for generation in range(self.generations):
        new_population = []
        current_best = min(population, key=self.calculate_fitness)
        current_best_fitness = self.calculate_fitness(current_best)

        if current_best_fitness < best_fitness:
            best_solution = current_best
            best_fitness = current_best_fitness

        self.best_fitness_history.append(best_fitness)
        new_population.append(current_best)

        while len(new_population) < self.population_size:
            parent1 = self.tournament_selection(population)
            parent2 = self.tournament_selection(population)
            child1, child2 = self.crossover(parent1, parent2)
            new_population.extend([self.mutate(child1), self.mutate(child2)])

        population = new_population[:self.population_size]
        print(f"Generation {generation}: Best fitness = {best_fitness}")

    return best_solution, best_fitness
```

### 4.3 Parametry algorytmu

Parametry algorytmu są przyjmowane w konstruktorze klasy implementującej rozwiązanie algorytmu.

```python
class GeneticTaskAllocation:
    """Genetic algorithm implementation for task allocation."""

    def __init__(self, tasks: List[Task], processors: List[Processor],
        population_size: int = 1000, generations: int = 5000,
        mutation_rate: float = 0.01, tournament_size: int = 3)-> None:
        """
        Initialize the genetic algorithm solver.
```

### 4.4 Obsługa danych wejściowych

Dane wejściowe są wprowadzane jako lista zadań oraz procesorów. Zadanie jest reprezentowane jako obiekt klasy Task, posiadający pola identyfikacji zadania oraz czasu egzekucji. Procesor jest reprezentowany jako obiekt klasy Processor, posiadający pola identyfikacji procesora oraz mnożnika.

```python
@dataclass
class Task:
    """Represents an task with execution time."""
    id: int
    execution_time: float

@dataclass
class Processor:
    """Represents a processor with multiplayer."""
    id: int
    multiplayer: float
```

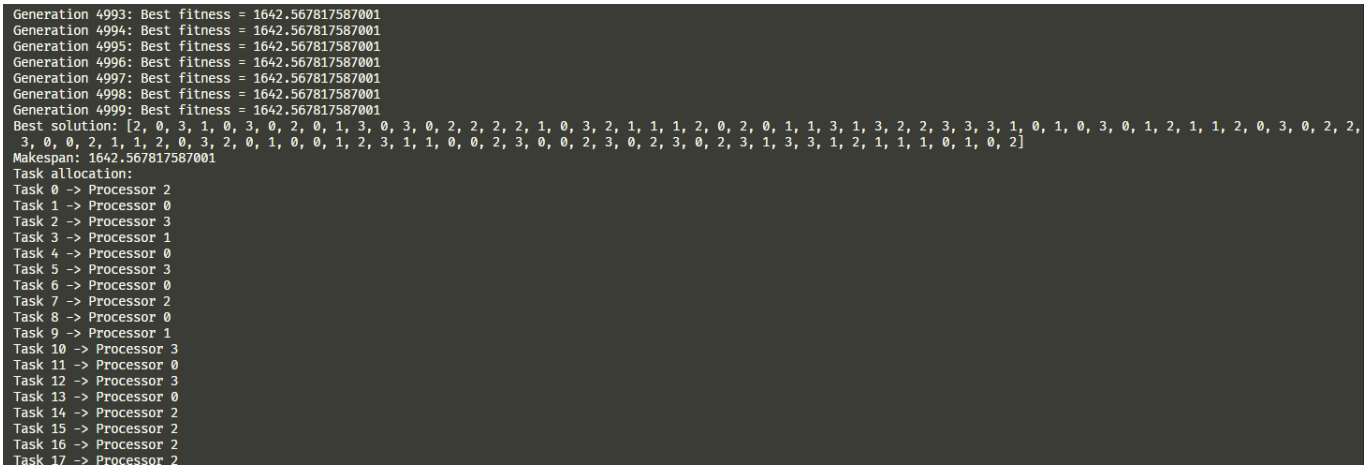
---

## 5. Wyniki

### 5.1 Wyniki działania programu

Przykład wyniku dla powyższego zestawu danych:

- Ilość zadań: 100
- Czas egzekucji zadań z zakresu 10 do 90
- Wielkość turnieju: 5
- Prawdopodobieństwo mutacji: 0.1
- Mnożniki procesorów: [1, 1.25, 1.5, 1.75]




*Rysunek 2 Wycinek działania programu w konsoli*

# Rozwiązanie problemu komiwojażera

---

### 1. Wprowadzenie

Problem komiwojażera to kolejny problem optymalizacyjny. Polega on na znalezieniu najkrótszej trasy, którą komiwojażer powinien pokonać, odwiedzając każdą z określonych miejscowości dokładnie raz i wracając na punkt początkowy. Celem jest minimalizacja całkowitej odległości lub kosztu podróży.

W kontekście algorytmu genetycznego, rozwiązanie problemu komiwojażera polega na reprezentacji trasy jako chromosomu, a następnie zastosowaniu operatorów by znaleźć najkrótszą trasę.

---

### 2. Opis teoretyczny algorytmu genetycznego

Algorytm genetyczny w tym przypadku będzie analogiczny do poprzednich problemów, ale ze specyficzną reprezentacją rozwiązań i zastosowaniem odpowiednich operatorów genetycznych, które uwzględniają strukturę trasy.

#### 2.1 Reprezentacja rozwiązań

Rozwiązanie problemu jest kodowane jako chromosom będący permutacją miejscowości, które komiwojażer ma odwiedzić. Każdy chromosom jest tablicą liczb całkowitych, w której każda liczba oznacza indeks miejscowości w danej trasie. Długość chromosomu jest równa liczbie miejscowości, a wartości w tablicy wskazują, w jakiej kolejności są odwiedzane. Na przykład, jeśli mamy pięć miejscowości, możliwy chromosom może wyglądać jak [2, 0, 3, 4, 1], co oznacza, że komiwojażer odwiedza miejscowości w tej właśnie kolejności.

#### 3.2 Populacja początkowa

Na początku generowana jest losowa populacja początkowa o ustalonej liczbie osobników.

#### 3.3 Funkcja dopasowania (fitness function)

Funkcja dopasowania ocenia jakość danego rozwiązania, obliczając całkowitą odległość pokonaną przez komiwojażera w danej trasie. Odległość ta jest sumą odległości między kolejnymi miejscowościami w trasie, uwzględniając zarówno odwiedzanie wszystkich miejscowości, jak i powrót do punktu początkowego. Funkcja fitness minimalizuje tę całkowitą odległość. Im krótsza trasa, tym lepsze rozwiązanie. Wartość fitness jest odwrotnością całkowitej odległości, więc im wyższa wartość fitness, tym lepsze rozwiązanie.

#### 3.4 Operatory genetyczne

7. **Selekcja**: Wyboru rodziców do krzyżowania dokonuje się na podstawie ich wartości fitness.
8. **Krzyżowanie (crossover)**: Dwa chromosomy rodziców są łączone w celu stworzenia potomków.
9. **Mutacja**: Każdy gen chromosomu może ulec losowej zmianie przydziału procesora z określonym prawdopodobieństwem.

#### 3.5 Kryteria zakończenia

Algorytm kończy działanie po osiągnięciu maksymalnej liczby iteracji lub braku poprawy najlepszego rozwiązania przez określoną liczbę generacji.

---

## 4. Opis implementacji programu

### 4.1 Język i środowisko

Do implementacji wykorzystano język Python. Część kodu została użyta ponownie z poprzednich problemów.

### 4.2 Struktura kodu

Program składa się z następujących modułów:

- **Generowanie populacji**: Funkcja inicjalizująca losową populację, w której każdy chromosom reprezentuje permutację miejscowości.

```python
def create_initial_population(self) -> List[List[int]]:
    """Create random initial population of chromosomes."""
    population = []
    for _ in range(self.population_size):
        chromosome = list(range(len(self.cities)))
        random.shuffle(chromosome)
        population.append(chromosome)
    return population
```

- **Funkcja dopasowania**: Oblicza wartość dopasowania dla każdego osobnika w populacji. Funkcja ta oblicza całkowitą odległość dla danej trasy i na jej podstawie określa wartość fitness. Im krótsza trasa, tym wyższa wartość fitness.

```python
def calculate_fitness(self, chromosome: List[int]) -> float:
    """Calculate fitness value for a chromosome."""
    total_distance = 0.0
    for i in range(len(chromosome)):
        city1 = self.cities[chromosome[i]]
        city2 = self.cities[chromosome[(i + 1) % len(chromosome)]]
        total_distance += math.sqrt((city1.x - city2.x) ** 2 + (city1.y - city2.y) ** 2)
    return 1 / total_distance
```

- **Operatory genetyczne**: Implementacja selekcji metodą turniejową, krzyżowania metodą z zachowaniem porządku OX oraz losowej mutacji genów przy pomocy inwersji.

```python
def tournament_selection(self, population: List[List[int]]) -> List[int]:
    """Select chromosome using tournament selection."""
    tournament = random.sample(population, self.tournament_size)
    return max(tournament, key=self.calculate_fitness)

def crossover(self, parent1: List[int], parent2: List[int]) -> Tuple[List[int], List[int]]:
    """Perform ordered crossover between parents."""
    start, end = sorted(random.sample(range(len(parent1)), 2))
    child1 = [None] * len(parent1)
    child2 = [None] * len(parent2)

    child1[start:end] = parent1[start:end]
    child2[start:end] = parent2[start:end]

    fill_child(child1, parent2, end)
    fill_child(child2, parent1, end)

    return child1, child2

def mutate(self, chromosome: List[int]) -> List[int]:
    """Apply inversion to chromosome."""
    if random.random() < self.mutation_rate:
        i, j = random.sample(range(len(chromosome)), 2)
        chromosome[i], chromosome[j] = chromosome[j], chromosome[i]
    return chromosome
```

```python
def fill_child(child: List[int], parent: List[int], end: int) -> None:
    """Helper function to fill the child chromosome with remaining genes from the parent."""
    current_pos = end
    for gene in parent:
        if gene not in child:
            if current_pos >= len(child):
                current_pos = 0
            child[current_pos] = gene
            current_pos += 1
```

- **Główna pętla algorytmu**: Obsługuje proces ewolucji i zapisuje wyniki.

  **Schemat działania**:

  1. Inicjacja – losowa generacja populacji początkowej

  2. Tworzenie kolejnych pokoleń dopóki nie zostanie osiągnięty warunek stopu (z góry zdefiniowana liczba pokoleń):

     g) Obliczenie funkcji przystosowania.
     h) Przeprowadzenie selekcji.
     i) Zastosowanie operatorów genetycznych.

```python
def solve(self) -> Tuple[List[int], float]:
    """..."""
    population = self.create_initial_population()
    best_solution = None
    best_fitness = 0

    for generation in range(self.generations):
        new_population = []
        current_best = max(population, key=self.calculate_fitness)
        current_best_fitness = self.calculate_fitness(current_best)

        if current_best_fitness > best_fitness:
            best_solution = current_best
            best_fitness = current_best_fitness

        self.best_fitness_history.append(best_fitness)
        new_population.append(current_best)

        while len(new_population) < self.population_size:
            parent1 = self.tournament_selection(population)
            parent2 = self.tournament_selection(population)
            child1, child2 = self.crossover(parent1, parent2)
            new_population.extend([self.mutate(child1), self.mutate(child2)])

        population = new_population[:self.population_size]
        logging.info(f"Generation {generation}: Best fitness = {best_fitness}")

    return best_solution, best_fitness
```

### 4.3 Parametry algorytmu

Parametry algorytmu są przyjmowane w konstruktorze klasy implementującej rozwiązanie algorytmu.

```python
class GeneticTSP:
    """Genetic algorithm implementation for solving the Traveling Salesman Problem."""

    def __init__(self, cities: List[City],
                 population_size: int = 100, generations: int = 1000,
                 mutation_rate: float = 0.1, tournament_size: int = 5) -> None:
        """
```

### 4.4 Obsługa danych wejściowych

Dane wejściowe zawierają listę miejscowości (punktów, które należy odwiedzić). Każda miejscowość jest reprezentowana jako obiekt klasy **City**, który posiada pola identyfikacji i współrzędne

geograficzne. Na podstawie tych danych obliczana jest odległość pomiędzy poszczególnymi punktami.

```python
@dataclass
class City:
    """Represents a city with x and y coordinates."""
    x: float
    y: float
```

---

**5. Wyniki**

**5.1 Wyniki działania programu**

Przykład wyniku dla powyższego zestawu danych:

- Ilość miast: 100
- Koordynaty miast losowane z zakresu 10 do 90
- Wielkość turnieju: 5
- Prawdopodobieństwo mutacji: 0.1

```
INFO:root:Generation 993: Best fitness = 0.0009872917168971206
INFO:root:Generation 994: Best fitness = 0.0009872917168971206
INFO:root:Generation 995: Best fitness = 0.0009893156828275265
INFO:root:Generation 996: Best fitness = 0.0009893156828275265
INFO:root:Generation 997: Best fitness = 0.0009902667429344052
INFO:root:Generation 998: Best fitness = 0.0009902667429344052
INFO:root:Generation 999: Best fitness = 0.0009902667429344052
Best solution: [54, 83, 92, 22, 45, 33, 58, 35, 67, 87, 2, 39, 32, 76, 66, 97, 4, 49, 93, 86, 61, 55, 95, 12, 11, 53, 60, 50, 78, 73, 27, 25, 24, 80, 31, 10, 59, 38, 3, 14, 64, 96,
 82, 77, 44, 99, 79, 43, 26, 21, 75, 1, 90, 52, 84, 81, 29, 0, 20, 23, 69, 94, 30, 68, 85, 46, 70, 28, 6, 41, 48, 56, 13, 18, 9, 72, 89, 36, 65, 16, 62, 88, 40, 74, 5, 37, 34, 71,
98, 19, 15, 7, 57, 17, 63, 42, 51, 91, 47, 8]
Total distance: 1009.8289245145734
City order:
City 54: City(x=55.0164067064436, y=66.03614947848271)
City 83: City(x=46.69905811593357, y=50.606345336131966)
City 92: City(x=41.132231636679876, y=43.290427526550914)
City 22: City(x=40.525111671778355, y=43.55497117670002)
```

*Rysunek 2 Wycinek działania programu w konsoli*

# Podsumowanie

Zrealizowane implementacje stanowią solidny fundament do dalszych badań i ulepszeń algorytmów genetycznych, zwłaszcza w zakresie wydajności i adaptacyjności. Możliwe kierunki rozwoju to: równoległe obliczenia, dynamiczna adaptacja parametrów oraz zastosowanie alternatywnych metod selekcji i krzyżowania.