

Analiza Projektu: Algorytm Genetyczny dla Problemu Plecakowego

1. Objasnienie Struktury Kodu

`run_experiments.py` - Panel Sterowania

To jest główny plik, który uruchamia całe doświadczenie. Nie zawiera on logiki samego algorytmu, ale działa jak "Panel Sterowania" lub scenariusz eksperymentu.

- Definiuje, które pliki z danymi mają zostać przetestowane (lista DATA_FILES).
- Definiuje zestawy parametrów dla różnych eksperymentów (np. scenarios_rates, scenarios_selection_5_0). To tutaj decydujemy, jakie metody chcemy ze sobą porównać.
- Zarządza pętlą, która dla każdego pliku z danymi uruchamia po kolej wszystkie zdefiniowane scenariusze.

`main_experiment.py` - Laboratorium

Ten plik działa jak pojedyncze "Laboratorium". Jego zadaniem jest przeprowadzenie jednego pełnego eksperymentu dla jednego pliku z danymi.

- Funkcja run_and_collect_results przyjmuje dane problemu oraz listę scenariuszy (np. 3 różne ustawienia mutacji).
- Przeprowadza ewolucję dla każdego scenariusza, zbierając historię najlepszych wyników z każdego pokolenia.
- Funkcja report_results_and_plot bierze zebrane wyniki i generuje z nich końcowy wykres oraz tabelę w konsoli.

`ga_core.py` - Silnik Ewolucji

To jest serce i silnik całego programu. Zawiera kluczowe mechanizmy potrzebne do przeprowadzenia ewolucji.

- **import_data:** Funkcja odpowiedzialna za wczytanie pliku tekstowego (np. f1_1-d...) i przetłumaczenie go na format zrozumiały dla programu (pojemność plecaka i lista przedmiotów).

- **genetic_algorithm:** To jest "serce" ewolucji. Ta funkcja:
 - Tworzy losową "populację" plecaków.
 - Rozpoczyna pętlę "pokoleń".
 - Ocenia każdy plecak (jak dobry jest?).
 - Zapisuje najlepszy wynik z danego pokolenia.
 - Wybiera "rodziców" (używając np. selekcji ruletkowej).
 - "Krzyżuje" rodziców, by stworzyć "dzieci" (nową generację plecaków).
 - "Mutuje" niektóre dzieci (wprowadza losowe zmiany).
 - Zastępuje starą populację nową i powtarza proces.
 - Na końcu zwraca najlepszy plecak, jaki kiedykolwiek znalazł.

data_models.py - Słownik / Definicje

Ten plik niczego nie liczy. Działa jak słownik, który definiuje podstawowe "klocki" (obiekty), z których zbudowany jest problem.

- **Item (Przedmiot):** Definiuje, że przedmiot ma wartość i wagę.
- **KnapsackProblem (Problem Plecakowy):** Definiuje, że problem składa się z pojemności plecaka oraz listy przedmiotów.
- **Individual (Osobnik / Plecak):** Definiuje pojedyncze rozwiązanie. Jego najważniejszą częścią jest chromosome – lista zer i jedynek, która mówi, czy dany przedmiot jest w plecaku (1) czy go nie ma (0).

operators.py (Plik zakładany) - Zasady Ewolucji

Ten plik (którego nie analizowałem, ale którego działanie jest widoczne) zawiera "Zasady Gry" ewolucyjnej, czyli konkretne mechanizmy działania.

- **calculate_fitness (Funkcja Przystosowania):** Kluczowa funkcja. Ocenia, jak dobry jest dany plecak (Osobnik). Sumuje wartość wszystkich przedmiotów (gdzie gen to '1'). Co najważniejsze, jeśli suma wag przekroczy pojemność plecaka, funkcja zwraca **wartość 0**. To jest "kara" za spakowanie za ciężkiego plecaka.
- **Funkcje Selekcji (roulette..., ranking..., tournament...):** To są trzy różne metody wybierania "rodziców" do rozmnażania.
- **Funkcje Krzyżowania (single_point..., two_point...):** Dwie różne metody "mieszania genów" rodziców, by stworzyć potomstwo.
- **mutation:** Mechanizm losowej zmiany jednego "genu" (bitu) z 0 na 1 lub z 1 na 0.

2. Analiza Wyników Eksperymentalnych

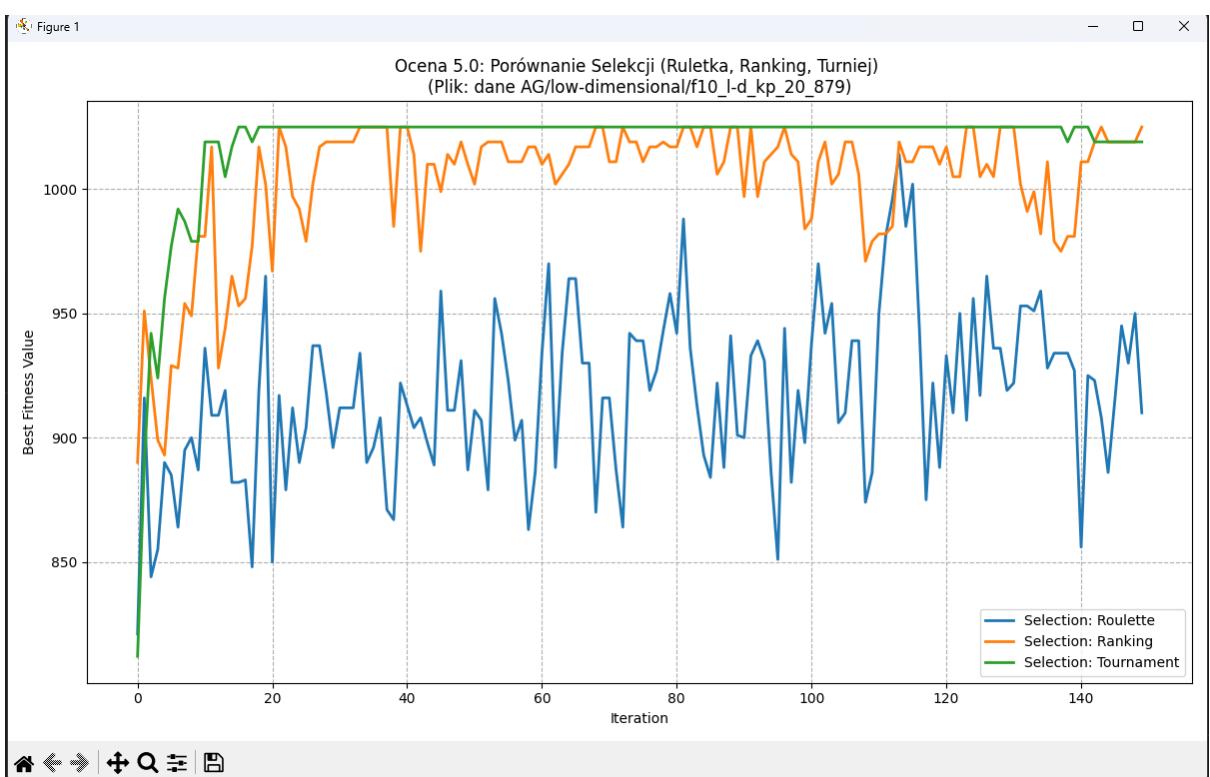
Na podstawie wygenerowanych wykresów można wyciągnąć kluczowe wnioski na temat działania algorytmu genetycznego. Eksperymenty ujawniły dwa drastycznie różne zachowania algorytmu w zależności od skali problemu.

2.1. Sukces ewolucji na danych low-dimensional

Wykresy dla zbiorów danych low-dimensional (np. f1_1-d..., f2_1-d..., f10_1-d...) pokazują, że algorytm genetyczny działa poprawnie i skutecznie znajduje rozwiązania.

- **Wpływ selekcji**

np. Wykres:



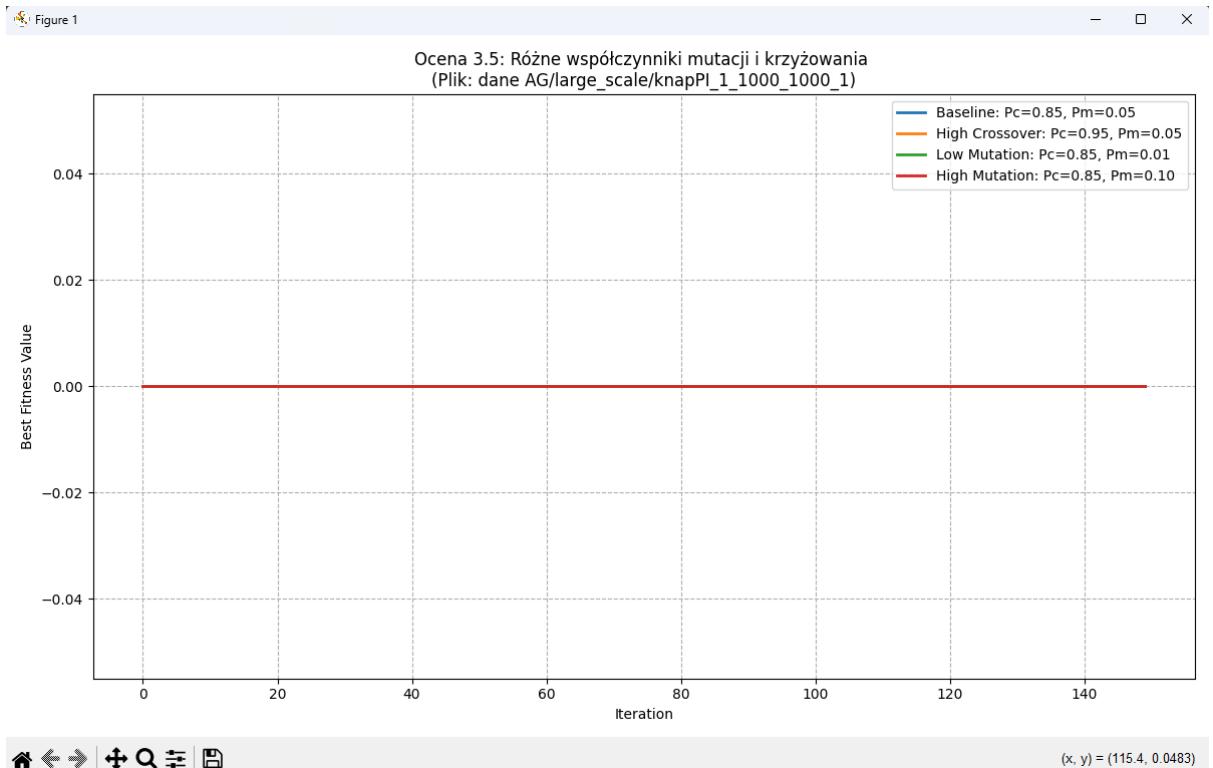
- **Selekcja Ruletkowa (niebieska linia)** jest najmniej stabilna. Działa bardziej losowo, co pozwala słabszym osobnikom przetrwać. Prowadzi to do dużych wahań ("szumu") i często wolniejszego dochodzenia do optymalnego wyniku.
- **Selekcja Rankingowa (pomarańczowa) i Turniejowa (zielona)** są wyraźnie lepsze. Obie metody są bardziej "elitarne" – silnie faworyzują najlepsze osobniki. Powoduje to silną presję selekcyjną, co skutkuje:
 - **Szybką zbieżnością:** Algorytm bardzo szybko (w ciągu ok. 20 pokoleń) znajduje rozwiązanie bliskie optimum.

- **Stabilnością:** Po znalezieniu dobrego wyniku, algorytm "trzyma się" go, a wykres się wypłaszcza na wysokim poziomie.
- **Wpływ krzyżowania (np. `image_c1760a.png`):**
 - Różnice między krzyżowaniem 1-punktowym a 2-punktowym są niewielkie. Obie metody wykazują podobną, niestabilną charakterystykę (co jest typowe przy selekcji ruletkowej, która była tu bazą). Obie metody skutecznie rekombinują materiał genetyczny.

2.2. Porażka ewolucji na danych `large_scale`

Wykresy dla zbiorów `large_scale` (np. `knapPI_1_1000_1000_1` czy `knapPI_1_2000_1000_1`) pokazują zupełnie inny obraz.

- **Obserwacja (np.):**



- Wszystkie linie na wykresach są idealnie płaskie i mają wartość `Best Fitness Value = 0.00` przez wszystkie 150 iteracji. Nie ma znaczenia, czy zmienialiśmy współczynniki mutacji, czy metody selekcji – wynik jest zawsze ten sam.
- **Diagnoza Problemu:**
 - Zgodnie z założeniami projektu, funkcja przystosowania (fitness) przyznaje **wartość 0** każdemu plecakowi, który przekroczy dozwoloną wagę (pojemność).

- Płaska linia na poziomie 0 oznacza, że przez 150 pokoleń algorytm **ani razu nie znalazł ani jednego plecaka**, który *nie* przekraczałby pojemności.
- Przyczyną jest **inicjalizacja populacji**. Funkcja `generate_random_individual` (z `data_models.py`) tworzy chromosom, losując 0 lub 1 z równym prawdopodobieństwem (50%).
- Dla problemu z 1000 przedmiotów (`knapPI_1_1000...`), oznacza to, że pierwszy "losowy" plecak zawiera średnio **500 przedmiotów**. Jest to o wiele za dużo, co gwarantuje przekroczenie wagi i fitness równy 0.
- Ponieważ **cała** populacja początkowa składa się z osobników o wartości 0, mechanizmy selekcji (Ruletka, Ranking, Turniej) zawodzą. Nie da się wybrać "lepszego" osobnika, skoro **wszystkie są jednakowo bezwartościowe**. Ewolucja nie ma jak się rozpocząć.

2.3. Sugerowane Rozwiązania Problemu `large_scale`

Aby algorytm mógł poradzić sobie z dużymi problemami, konieczna jest zmiana strategii:

1. **Modyfikacja inicjalizacji:** Najprostszym rozwiązaniem jest zmiana funkcji `generate_random_individual` tak, by tworzyła "rzadsze" plecaki. Zamiast losować 0/1 z szansą 50/50, można ustawić znacznie mniejsze prawdopodobieństwo wzięcia przedmiotu, np. 5% lub 1%. To dałoby algorytmowi szansę na znalezienie prawidłowych rozwiązań już na starcie.
2. **Wprowadzenie "Naprawy" Osobników (Repair Operator):** Lepszym podejściem jest dodanie funkcji "naprawczej". Po krzyżowaniu i mutacji, jeśli osobnik jest za ciężki, funkcja ta w pętli usuwałaby z niego losowe przedmioty, aż jego waga spadnie poniżej limitu. To gwarantuje, że *każdy* osobnik w populacji jest "legalny", a funkcja fitness może je poprawnie oceniać i porównywać.

3. Linijki kodu do nauczenia się

Objaśnienie kluczowych fragmentów kodu, podzielone według plików.

`run_experiments.py` (Twój Panel Sterowania)

Ten plik jest "głównym włącznikiem" i "panelem sterowania" dla całego projektu. Jego zadaniem jest **definiowanie, co chcesz przetestować**.

Python

```
# --- Plik: run_experiments.py ---

# Lista "pacjentów" do przebadania.
DATA_FILES = [
    "dane AG/low-dimensional/f1_l-d_kp_10_269",
    # ... i inne pliki ...
    "dane AG/large_scale/knapsackPI_1_1000_1000_1",
]

# "Domyślna recepta" dla każdego eksperymentu.
# Jeśli scenariusz nie powie inaczej, użyj tych ustawień.
BASE_PARAMS = {
    'pop_size': 100,
    'generations': 150,
    'p_crossover': 0.85,
    'p_mutation': 0.05,
    'selection_func': roulette_wheel_selection,
    'crossover_func': single_point_crossover
}

# --- Scenariusze dla Oceny 5.8 ---
# To jest serce Twoich eksperymentów.
# Tworzyesz listę "zadań do wykonania" dla laboratorium.
scenarios_selection_5_8 = [
    {
        # Zadanie 1: "Użyj 'Ruletki'"
        'label': 'Selection: Roulette',
        'selection_func': roulette_wheel_selection,
        # ...wciąż resztę ustawień z "domyślnej recepty" (BASE_PARAMS)
        **{k: v for k, v in BASE_PARAMS.items() if k not in ['selection_func']}

    },
    {
        # Zadanie 2: "Użyj 'Rankingu'"
        'label': 'Selection: Ranking',
        'selection_func': ranking_selection,
        # ...wciąż resztę ustawień z "domyślnej recepty"
        **{k: v for k, v in BASE_PARAMS.items() if k not in ['selection_func']}
    },
    # ... i tak dalej dla Turnieju ...
]

# --- PĘTLA WYKONUJĄCA EKSPERIMENTY ---

# Główna pętla programu.
# "Wciąż każdego 'pacjenta' (plik) z listy DATA_FILES po kolei..."
for filename in DATA_FILES:

    # ... i wyślij go do 'laboratorium' (funkcji ponizej)
    # razem z zestawem zadań 'Ocena 5.8'.
    run_experiments_comparisons(
        filename,
        scenarios_selection_5_8,
        experiment_title="Ocena 5.8: Porównanie Selekcji"
    )
    # (Tutaj uruchamiasz też inne eksperymenty 3.5, 4.5 itd.)
```

main_experiment.py (Laboratorium)

Ten plik to "laboratorium". Otrzymuje "pacjenta" (dane) i "listę zadań" (scenariusze) z pliku `run_experiments.py`, a następnie wykonuje całą ciężką pracę.

Python

```
# --- Plik: main_experiment.py ---

def run_and_collect_results(
    problem_data: KnapsackProblem,
    scenarios: list
) -> list[dict]:
    """
    To jest 'Procedura Badawcza' w Twoim laboratorium.
    Bierze problem (pacjenta) i listę zadań (scenariuszy).
    """

    results = []

    # "Dla każdego zadania na liście (np. 'testuj Ruletką', 'testuj Ranking')...""
    for scenario in scenarios:

        # ...URUCHOM GŁÓWNY SILNIK EVOLUCJI.
        # To jest ten "włącznik" algorytmu genetycznego.
        # Wciśnij "START" i czekaj na wyniki.
        best_individual, fitness_history = genetic_algorithm(
            problem=problem_data,
            population_size=scenario['pop_size'],
            generations=scenario['generations'],
            # ...przekaż wszystkie inne ustawienia ze scenariusza...
            selection_func=scenario['selection_func'],
            crossover_func=scenario['crossover_func']
        )

        # "Zapisz wyniki do 'tarczki pacjenta'."
        # Najważniejsze to 'fitness_history' - to są dane na wykres.
        results.append({
            'label': scenario['label'], # Etykieta (np. "Selection: Roulette")
            'fitness_history': fitness_history, # Lista wyników (np. [100, 120, 150])
        })

    # "Zwróć wypełnioną 'tarczkę' z wynikami."
    return results

def report_results_and_plot(experiment_name: str, results: list[dict]):
    """
    Bierze 'tarczkę' z wynikami i ją publikuje (rysuje wykres).
    """

    # ... (pomijam drukowanie w konsoli) ...

    # "Dla każdego wyniku w 'tarczce'..."
    for r in results:
        # TO JEST KLUCZOWA LINIA:
        # "Narysuj linię na wykresie używając historii fitness
        # i nadaj jej etykietę, którą zdefiniowaliśmy wcześniej"
        plt.plot(r['fitness_history'], label=r['label'], linewidth=2)

    # ... (ustawiania wykresu: tytuł, siatka, legenda) ...

    # "Pokaż gotowy wykres na ekranie."
    plt.show()
```

ga_core.py (Silnik Ewolucji)

To jest serce programu. Tutaj dzieje się cała "magia" ewolucji.

Python

```
# --- Plik: ga_core.py ---

def import_data(file_name: str) -> KnapsackProblem:
    """
    Wczytuje plik tekstowy z danymi i tłumaczy go na
    obiekty 'KnapsackProblem', które reszta programu rozumie.
    To jest funkcja, którą poprawialiśmy, by czytała oba formaty danych.
    """
    # ... (logika wczytywania i parsowania pliku) ...

    # Tworzy i zwraca "problem"
    return KnapsackProblem(capacity, items)

# -----
# --- NAJWAŻNIEJSZA FUNKCJA W PROJEKCIE ---
# -
def genetic_algorithm(
    # ... (przyjmuje wszystkie parametry: problem, pop_size, p_crossover itd.) ...
) -> tuple[Individual, list[float]]:
    """
    Główny silnik ewolucji.
    """

    # Przechowuje najlepszy wynik z KAŻDEGO pokolenia (do wykresu)
    fitness_history = []

    # KROK 1: INICJALIZACJA
    # Stwórz pierwszą, zupełnie losową populację "plecaków"
    population = [generate_random_individual(chromosome_length) for _ in range(population_size)]

    # KROK 2: PĘTLA POKOLEŃ (Ewolucja)
    # "Powtórz cały proces ewolucji 'generations' razy (np. 150)"
    for gen in range(generations):

        # KROK 2a: OCENA (FITNESS)
        # "Oceń każdy plecak w populacji: Jak jest dobry?"
        for individual in population:
            # (Ta funkcja jest w "operators.py", ale tu jest wywoływana)
            # Oblicza wartość i wagę, a potem zapisuje wynik w individual.fitness
            calculate_fitness(individual, problem)

        # KROK 2b: ZAPIS HISTORII
        # "Znajdź 'mistrza' tego pokolenia (plecak z najwyższym fitnessem)..."
        best_individual = max(population, key=lambda i: i.fitness)
        # "...i zapisz jego wynik. To jest punkt na Twoim wykresie!"
        fitness_history.append(best_individual.fitness)

        # KROK 2c: TWORZENIE NOWEJ POPULACJI
        new_population = []
        while len(new_population) < population_size:

            # KROK 2c-1: SELEKCJA
            # "Wybierz rodziców" (używając funkcji, którą podałeś,
            # np. roulette_wheel_selection)
            parent1 = selection_func(population)
            parent2 = selection_func(population)

            # KROK 2c-2: KRZYŻOWANIE (ROZMNĄZANIE)
            # "Czy rodzice będą mieli 'dzieci' (krzyżowanie)?"
            if random.random() < crossover_probability:
                # Tak -> stwórz "dzieci" mieszając geny rodziców
                offspring1, offspring2 = crossover_func(parent1, parent2)
            else:
                # Nie -> "dzieci" są klonami rodziców
                offspring1, offspring2 = Individual(parent1.chromosome[:]), Individual(parent2.chromosome[:])

            # KROK 2c-3: MUTACJA
            # "Wprowadź losowe zmiany (mutacje) u dzieci"
            offspring1.mutate(mutation_rate)
            offspring2.mutate(mutation_rate)

            # Dodaj nowe osobniki do nowej populacji
            new_population.append(offspring1)
            new_population.append(offspring2)

    # KROK 3: WYKRESZENIE
    # "Wyświetl wykres fitness_history"
    plot(fitness_history)
```

```
# Dodaj nowe "dzieci" do nowej populacji"
new_population.append(offspring1)
new_population.append(offspring2) # (Kod sprawdza czy jest miejsce)

# KROK 2d: WYMIANA POKOLEŃ
# "Starą populację "umiera". Nowa (dzieci) zajmuje jej miejsce."
population = new_population

# KROK 3: ZAKOŃCZENIE
# Po wszystkich pokoleniach, ocen ostatnią populację...
for individual in population:
    calculate_fitness(individual, problem)
# ...i znajdź absolutnie najlepszy pląsak, jaki kiedykolwiek powstał.
final_best_individual = max(population, key=lambda i: i.fitness)

# Zwróć najlepszy pląsak ORAZ historię wyników na wykres.
return final_best_individual, fitness_history
```

data_models.py (Definicje / Słownik)

Ten plik to "słownik". Nie wykonuje żadnych skomplikowanych operacji, ale **definiuje "klocki"**, z których zbudowany jest cały program.

Python



```
# --- Plik: data_models.py ---\n\n\nclass Item:\n    """Definiuje, czym jest 'Przedmiot'.\"\n    def __init__(self, index, value, weight):\n        self.value = value # Ma wartość\n        self.weight = weight # Ma wagę\n\n\nclass KnapsackProblem:\n    """Definiuje, czym jest 'Problem'.\"\n    def __init__(self, capacity, items):\n        self.capacity = capacity # Ma pojemność\n        self.items = items # Ma listę przedmiotów\n\n\nclass Individual:\n    """\n    Definiuje, czym jest 'Osobnik' (czyli pojedyncze rozwiązanie / plecak).\n    TO JEST BARDZO WAŻNA KLASA.\n    """\n    def __init__(self, chromosome: list[int]):\n        # "DNA" OSOBNIKA.\n        # Lista zer i jedynek, np. [0, 1, 1, 0]\n        # '1' oznacza "weź przedmiot", '0' oznacza "zostaw"\n        self.chromosome = chromosome\n\n        # "SIŁA" / OCENA OSOBNIKA.\n        # Jaką ma łączną wartość? Wypełniane przez 'calculate_fitness'.\n        self.fitness = 0.0\n\n        # łączna waga. Potrzebna do sprawdzenia, czy plecak nie jest za ciężki.\n        self.total_weight = 0\n\n    def generate_random_individual(length: int) -> Individual:\n        """\n        Tworzy jeden, zupełnie losowy plecak.\n        To ta funkcja, która losowała 50/50 i powodowała\n        płaskie wykresy na 0.0 dla dużych problemów.\n        """\n        chromosome = [random.randint(0, 1) for _ in range(length)]\n        return Individual(chromosome)
```

operators.py (Zasady Ewolucji - Zakładane)

absolutnie kluczowe funkcje, aby projekt działał

Python



```
# --- Plik: operators.py (Zakładany) ---\n\ndef calculate_fitness(individual: Individual, problem: KnapsackProblem):\n    """\n        NAJWAŻNIEJSZA ZASADA GRY. Ocenia, jak dobry jest plecak.\n    """\n\n    total_value = 0\n    total_weight = 0\n\n    # Przejdz przez "DNA" plecaka...\n    for i, gene in enumerate(individual.chromosome):\n        if gene == 1:\n            # Jeśli gen to '1', dodaj przedmiot do plecaka\n            total_value += problem.items[i].value\n            total_weight += problem.items[i].weight\n\n    # [cite_start]KARA ZA PRZEPEŁNIENIE (Kluczowy wymóg z PDF [cite: 135])\n    if total_weight > problem.capacity:\n        # Jeśli plecak jest za ciężki, jest BEZUŻYTECZNY.\n        individual.fitness = 0.0\n    else:\n        # Jeśli się mieści, jego "siła" to jego wartość.\n        individual.fitness = total_value\n\n    individual.total_weight = total_weight\n\n\ndef mutation(individual: Individual, mutation_probability: float):\n    """\n        [cite_start]Wprowadza losowe zmiany w "DNA" [cite: 128-130].\n    """\n\n    for i in range(len(individual.chromosome)):\n        # "Dla każdego genu jest mała szansa (np. 5%)..." \n        if random.random() < mutation_probability:\n            # "...że zostanie 'zmutowany' (zmieniony na przeciwny)"\n            individual.chromosome[i] = 1 - individual.chromosome[i] # (zamienia 0-1)\n\n\ndef single_point_crossover(parent1: Individual, parent2: Individual) -> tuple[Individual]:\n    """\n        [cite_start]"Miesza geny" rodziców w jednym punkcie [cite: 116-117, 120].\n    """\n\n    # ... (logika losowania punktu 'cut' i zamiany "ogonów") ...\n    return offspring1, offspring2\n\n# ... oraz funkcje selekcji (roulette, ranking, tournament) ...
```