Algorytmy ewolucyjne

Zasada działania, obszary zastosowań, biblioteki programistyczne, zastosowanie do wybranego problemu optymalizacyjnego oraz eksperymenty

Rafał Dubiel, Daniel Antolak

WSZiB, 2023

---

1. Zasada działania i zastosowania

1.1. Zasada działania algorytmów ewolucyjnych

Algorytmy ewolucyjne *(EA, Evolutionary Algorithms)* to klasa metod optymalizacyjnych, które bazują na mechanizmach inspirowanych teorią ewolucji biologicznej. Kluczowymi elementami algorytmów ewolucyjnych są populacja potencjalnych rozwiązań, selekcja najsilniejszych osobników, mutacja i rekombinacja, które pozwalają na przeszukiwanie przestrzeni rozwiązań problemu. *[1]*

Proces ewolucji w algorytmach tego typu składa się z kilku etapów:

- **Inicjalizacja populacji:** Na początku tworzy się populację składającą się z losowych rozwiązań problemu.
- **Ocena:** Każdy osobnik (rozwiązanie) jest oceniany przy użyciu funkcji dopasowania (fitness function), która mierzy, jak dobrze dany osobnik rozwiązuje problem.
- **Selekcja:** Najlepsze osobniki są wybierane do procesu reprodukcji, zgodnie z zasadą przetrwania najsilniejszych.
- **Rekombinacja i mutacja:** Wybrane osobniki krzyżują się, a ich "potomstwo" jest dodatkowo modyfikowane poprzez mutacje, co wprowadza różnorodność genetyczną w populacji.
- **Zastąpienie:** Nowa generacja osobników zastępuje całość lub część starej populacji. Proces ten jest powtarzany, aż do osiągnięcia kryteriów stopu (np. liczba pokoleń lub znalezienie rozwiązania o odpowiedniej jakości).

Istnieją dwie główne strategie ewolucyjne (μ, β) i $(\mu + \beta)$, gdzie μ rodziców produkuje β potomków. W (μ, β) najlepszych β potomków przeżywa i zastępuje rodziców. W rezultacie rodzice nie są obecni w następnej populacji. Przeciwnie, $(\mu + \beta)$ dopuszcza przeżycie zarówno potomków jak i rodziców. W $(\mu + \beta)$ stosowana jest strategia elitarna (najlepszy osobnik zawsze jest kopiowany do następnej populacji). *[1, 3]*

Jedną z kluczowych zalet algorytmów ewolucyjnych jest ich elastyczność i wszechstronność. Są one nie tylko niezawodne, ale również łatwe do adaptacji w przypadku różnych problemów optymalizacyjnych. Dodatkowym atutem jest brak konieczności posiadania wstępnej wiedzy o danym problemie – algorytmy te opierają się wyłącznie na funkcji celu, co czyni je wyjątkowo skutecznym narzędziem do przeszukiwania złożonych, wielowymiarowych przestrzeni rozwiązań. *[2]*

Algorytmy ewolucyjne mają również pewne ograniczenia. Jednym z głównych jest brak gwarancji znalezienia optymalnego rozwiązania w określonym czasie. Proces ewolucyjny może wymagać wielu iteracji, zanim osiągnie się optymalne rozwiązanie, a czas potrzebny na jego znalezienie bywa trudny do przewidzenia. Dodatkowo, skuteczność algorytmów ewolucyjnych silnie zależy od wartości parametrów, takich jak rozmiar populacji, prawdopodobieństwo mutacji czy współczynnik krzyżowania. Dobór właściwych wartości tych parametrów bywa wyzwaniem i często wymaga eksperymentalnego dostrajania. *[2]*

1.2. Zastosowania algorytmów ewolucyjnych

Algorytmy ewolucyjne znajdują zastosowanie w wielu różnych dziedzinach, zarówno w optymalizacji technicznej, jak i w zagadnieniach natury bardziej teoretycznej.

Przykładowe obszary zastosowań obejmują:

Optymalizacja inżynierska: Algorytmy ewolucyjne są wykorzystywane w projektowaniu systemów mechanicznych, elektroniki, konstrukcji budowlanych, gdzie trzeba optymalizować złożone struktury, których analiza może być trudna do wykonania klasycznymi metodami. *[1]*

Sztuczna inteligencja i uczenie maszynowe: Algorytmy ewolucyjne są stosowane do optymalizacji hiperparametrów modeli uczenia maszynowego, a także do automatycznego generowania struktur neuronowych. *[2]*

Bioinformatyka: Algorytmy ewolucyjne są używane do analizy sekwencji DNA, projektowania leków oraz modelowania procesów biologicznych, gdzie kluczowa jest zdolność algorytmu do eksploracji dużych i złożonych przestrzeni rozwiązań. *[2]*

Rozwiązywanie problemów kombinatorycznych: W zagadnieniach takich jak komiwojażer czy rozwiązywanie sudoku, algorytmy ewolucyjne mogą być użyte do znajdowania rozwiązań, które są trudne do uzyskania za pomocą innych metod. *[1]*

Optymalne zarządzanie: Algorytmy ewolucyjne są wykorzystywane w zarządzaniu w celu optymalizacji zasobów i procesów produkcyjnych. Mogą wspomagać tworzenie optymalnych harmonogramów i planów działań, uwzględniając takie czynniki jak dostępność zasobów, koszty oraz terminy. *[2, 3]*

Bibliografia

[1] Wykład nr.1 „Sztuczna inteligencja i systemy ekspertowe” – Rafał Dreżewski

[2] https://mfiles.pl/pl/index.php/Algorytm_ewolucyjny

[3] „Algorytmy ewolucyjne i ich zastosowania” – Ewa Figielska

---

2. Biblioteki programistyczne

Biblioteki **DEAP** i **Jenetics** są zaawansowanymi narzędziami służącymi do rozwiązywania złożonych problemów optymalizacyjnych. Umożliwiają eksperymentowanie z różnymi technikami ewolucyjnymi, takimi jak algorytmy genetyczne, programowanie genetyczne czy strategie ewolucyjne. Dzięki nim możliwe jest przeprowadzanie badań nad nowymi metodami optymalizacji, a także praktyczne zastosowanie tych technik w przemyśle, wspomagając optymalizację procesów, harmonogramowanie zadań czy projektowanie skomplikowanych systemów.

Biblioteka DEAP

(Distributed Evolutionary Algorithms in Python)

Otwarto-źródłowa biblioteka napisana w języku Python, która umożliwia łatwe tworzenie i testowanie algorytmów ewolucyjnych. DEAP pozwala użytkownikom na szybkie prototypowanie rozwiązań optymalizacyjnych oraz eksperymentowanie z różnymi podejściami.

Instalacja:

Możemy zainstalować bibliotekę DEAP np. przy pomocy menedżera pakietów pip, aby to zrobić wykonujemy komendę ***pip install deap*** w wierszu poleceń systemu operacyjnego. Wymagane jest posiadanie Pythona w wersji 2.7 lub 3.3 i nowszej oraz zainstalowanego menedżera pakietów pip. Można też pobrać bibliotekę DEAP bezpośrednio z kodu źródłowego klonując repozytorium przy pomocy komendy ***git clone https://github.com/DEAP/deap.git*** a następnie wykonując komendę ***python setup.py install***.

Możliwości:

DEAP oferuje szerokie możliwości, w tym elastyczne struktury danych pozwalające na definiowanie własnych reprezentacji osobników i populacji. Dostarcza gotowe operatory genetyczne, takie jak krzyżowanie, mutacja czy selekcja, oraz wsparcie dla programowania genetycznego, umożliwiając tworzenie programów jako drzew

składniowych. Biblioteka wspiera przetwarzanie równoległe, co pozwala na wykorzystanie wielu rdzeni procesora i przyspieszenie obliczeń. Dodatkowo, DEAP zawiera narzędzia statystyczne i analityczne do monitorowania i analizowania wyników ewolucji oraz integruje się z innymi bibliotekami, takimi jak NumPy czy SciPy.

W ramach DEAP można implementować i modyfikować różne algorytmy ewolucyjne, w tym algorytmy genetyczne, programowanie genetyczne, strategie ewolucyjne oraz ewolucję różnicową. Biblioteka umożliwia również wykorzystanie algorytmów wielokryterialnych, co pozwala na optymalizację wielu celów jednocześnie.

Zalety:

- **Modularność:** Łatwość dostosowywania i rozbudowy poszczególnych komponentów algorytmów.
- **Prostota użycia:** Przyjazny interfejs i czytelna składnia.
- **Wydajność:** Możliwość równoległego przetwarzania zwiększa szybkość obliczeń.
- **Społeczność:** Aktywne wsparcie i bogata dokumentacja.
- **Wszechstronność:** Możliwość zastosowania w różnych dziedzinach, od nauki po przemysł.

Biblioteka Jenetics

Zaawansowana biblioteka napisana w języku Java, przeznaczona do implementacji algorytmów genetycznych, strategii ewolucyjnych oraz programowania genetycznego. Wykorzystuje nowoczesne funkcje dostępne w Javie 8 i nowszych wersjach, takie jak strumienie i wyrażenia lambda, co pozwala na efektywne i czytelne tworzenie aplikacji ewolucyjnych.

Instalacja:

Aby zainstalować Jenetics w swoim projekcie Java, należy posiadać Java Development Kit (JDK) w wersji 8 lub nowszej oraz system budowania projektu, taki jak Maven lub Gradle. Instalacja polega na dodaniu odpowiedniej zależności do pliku konfiguracyjnego projektu. Po synchronizacji projektu biblioteka jest gotowa do użycia, a potrzebne klasy można zaimportować w kodzie Java.

Możliwości:

Jenetics oferuje szeroki zakres funkcjonalności, w tym generyczne reprezentacje genów i chromosomów, co ułatwia modelowanie problemów. Dostarcza bogaty zestaw operatorów genetycznych, takich jak krzyżowanie, mutacje czy selekcje, oraz zawiera silnik ewolucyjny, który umożliwia łatwe uruchamianie i kontrolowanie procesu ewolucji. Wykorzystanie programowania funkcyjnego pozwala na pisanie czytelnego i łatwego w utrzymaniu kodu.

W ramach Jenetics można implementować różne algorytmy ewolucyjne, w tym algorytmy genetyczne, strategie ewolucyjne i programowanie genetyczne. Dostępne są także algorytmy wielokryterialne oraz funkcje pozwalające na optymalizację z ograniczeniami.

Zalety:

- **Modularność i elastyczność:** Architektura Jenetics pozwala na łatwe rozszerzanie i dostosowywanie biblioteki do własnych potrzeb.
- **Wydajność:** Wykorzystanie nowoczesnych funkcji Javy zapewnia wysoką wydajność.
- **Czytelny kod:** Projektowanie oparte na zasadach programowania funkcyjnego ułatwia pisanie i utrzymanie kodu.
- **Dokumentacja i przykłady:** Dostępne są liczne przykłady oraz obszerna dokumentacja ułatwiająca naukę.
- **Społeczność i wsparcie:** Aktywna społeczność użytkowników i deweloperów.

---

3. Zaimplementowany algorytm rozwiązujący wybrany problem optymalizacyjny

Problemem który zdecydowaliśmy się rozwiązać przy użyciu algorytmów ewolucyjnych jest **Sudoku** - popularna gra logiczna polegająca na wypełnianiu kwadratowej planszy o wymiarach 9x9 cyframi od 1 do 9 w taki sposób, aby w każdym wierszu, każdej kolumnie oraz w każdym z dziewięciu podobszarów 3x3 cyfry się nie powtarzały.

Do stworzenia tego algorytmu wykorzystaliśmy wcześniej opisaną bibliotekę **DEAP** dla języka **python** ze względu na intuicyjny interfejs i prostotę użycia.

Poniższy program implementuje algorytm ewolucyjny mający na celu rozwiązywanie dowolnych plansz Sudoku. Problem jest rozwiązywany poprzez ewolucję populacji potencjalnych rozwiązań, gdzie osobniki są reprezentowane przez plansze Sudoku.

```
import numpy as np
import random
from deap import base, creator, tools, algorithms
```

Importujemy niezbędne narzędzia takie jak biblioteka **NumPy** która ułatwi operacje na planszach sudoku które są reprezentowane przez dwuwymiarowe tablice, biblioteka **random** która pozwoli na m.in. wypełnianie plansz sudoku losowymi wartościami oraz niezbędne narzędzia wymagane do implementacji algorytmów ewolucyjnych z biblioteki **DEAP**.

```
easy = [
    [5, 3, 0, 0, 7, 0, 9, 0, 0],
    [6, 7, 2, 1, 9, 0, 3, 4, 8],
    [0, 0, 8, 0, 0, 0, 5, 6, 0],
    [8, 0, 9, 0, 6, 1, 0, 2, 0],
    [4, 0, 6, 8, 5, 0, 7, 0, 0],
    [7, 1, 3, 0, 0, 0, 8, 5, 6],
    [9, 0, 1, 5, 0, 7, 2, 8, 0],
    [2, 0, 0, 0, 1, 0, 6, 0, 5],
    [0, 4, 5, 0, 8, 0, 0, 7, 0]
]

hard = [
    [5, 3, 0, 0, 7, 0, 0, 0, 0],
    [6, 0, 0, 1, 9, 5, 0, 0, 0],
    [0, 9, 8, 0, 0, 0, 0, 6, 0],
    [8, 0, 0, 0, 6, 0, 0, 0, 3],
    [4, 0, 0, 8, 0, 3, 0, 0, 1],
    [7, 0, 0, 0, 2, 0, 0, 0, 6],
    [0, 6, 0, 0, 0, 0, 2, 8, 0],
    [0, 0, 0, 4, 1, 9, 0, 0, 5],
    [0, 0, 0, 0, 8, 0, 0, 7, 9]
]

sudoku_board = np.array(easy)
```

Definiujemy dwuwymiarowe tablice które będą planszami sudoku do rozwiązania przez algorytm a następnie przekształcamy zdefiniowaną planszę na tablicę NumPy. Możemy wybrać czy algorytm ma rozwiązywać prostszą czy trudniejszą planszę.

```
creator.create("FitnessMin", base.Fitness, weights=(-1.0,))
creator.create("Individual", np.ndarray, fitness=creator.FitnessMin)
```

Definiuje klasę **FitnessMin**, która reprezentuje atrybut fitness, wykorzystywany do oceny osobników w algorytmie. Parametr **weights=(-1.0,)** oznacza, że algorytm będzie minimalizował funkcję celu. Druga linia natomiast tworzy klasę **Individual**, dziedziczącą po **np.ndarray**. Każdy obiekt tej klasy ma dodatkowy atrybut fitness, który jest instancją klasy **FitnessMin**.

```
def create_individual():

    individual = sudoku_board.copy()

    for i in range(9):
        row = individual[i]
        missing_numbers = list(set(range(1, 10)) - set(row))
        random.shuffle(missing_numbers)

        for j in range(9):
            if row[j] == 0:
                row[j] = missing_numbers.pop()

    return individual
```

Funkcja **create_individual()** generuje nowego osobnika (planszę Sudoku) na podstawie początkowej planszy **sudoku_board**.

Najpierw kopiujemy planszę dla tego osobnika, następnie dla każdego wiersza sprawdzamy jakich liczb jeszcze brakuje.

set(range(1,10)) tworzy nam zbiór wszystkich liczb 1-9 od którego odejmujemy zbiór liczb które znajdują się już w wierszu. Takim sposobem otrzymujemy listę elementów których brakuje w wierszu.

Brakujące liczby są w losowej kolejności wpisywane w puste komórki wiersza.

```
def eval_sudoku(individual):

    fitness = 0
```

```
    # sprawdzanie poprawności dla wierszy
    for row in individual:
        fitness += len(row) - len(set(row))

    # sprawdzanie poprawności dla kolumn
    for col in individual.T:
        fitness += len(col) - len(set(col))

    # sprawdzanie poprawności dla bloków
    for i in range(3):
        for j in range(3):
            block = individual[i*3:(i+1)*3, j*3:(j+1)*3].flatten()
            fitness += len(block) - len(set(block))

    return fitness,
```

Funkcja ewaluacyjna ocenia jak dobrym rozwiązaniem problemu jest dany osobnik.

Możemy założyć, że rozwiązanie planszy sudoku jest tym lepsze im mniej konfliktów występuje w wierszach, kolumnach oraz blokach 3x3. W związku z tym nakładamy karę zwiększając funkcję fitness za każdy konflikt w wierszu, kolumnie czy bloku.

```
def mutate_individual(individual):

    # komórki, które nie były zdefiniowane w początkowej planszy
    mutable_cells = [(i, j) for i in range(9) for j in range(9) if sudoku_board[i][j] == 0]

    # zamiana komórek miejscami
    if len(mutable_cells) >= 2:
        (x1, y1), (x2, y2) = random.sample(mutable_cells, 2)
        individual[x1, y1], individual[x2, y2] = individual[x2, y2], individual[x1, y1]

    return individual,
```

Ta funkcja odpowiedzialna jest za mutację osobników. Mutacja w tym wypadku polega na zamianie miejscami dwóch elementów, nie zdefiniowanych w początkowej planszy. **mutable_cells** są zapisane indeksy komórek które można mutować.

```
toolbox = base.Toolbox()
toolbox.register("individual", tools.initIterate, creator.Individual, create_individual)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluate", eval_sudoku)
toolbox.register("mate", tools.cxTwoPoint)
toolbox.register("mutate", mutate_individual)
toolbox.register("select", tools.selTournament, tournsize=3)
```

W tym fragmencie kodu definiujemy narzędzia potrzebne do działania algorytmu.

toolbox = base.Toolbox():

- Tworzy obiekt Toolbox, który służy jako kontener do rejestrowania funkcji.

toolbox.register("individual"…):

- Rejestruje funkcję do tworzenia pojedynczego osobnika (individual), który jest instancją klasy Individual, wypełnioną za pomocą funkcji create_individual.

toolbox.register("population"…):

- Rejestruje funkcję do tworzenia populacji, czyli listy osobników, na podstawie wcześniej zdefiniowanego generatora toolbox.individual.

toolbox.register("evaluate"…):

- Rejestruje funkcję oceniającą (evaluate), która oblicza wartość funkcji celu dla każdego osobnika, używając funkcji eval_sudoku.

toolbox.register("mate"…):

- Rejestruje operator krzyżowania dwupunktowego (mate), który wymienia fragmenty danych między dwoma osobnikami, tworząc nowe potomstwo.

toolbox.register("mutate"…):

- Rejestruje operator mutacji (mutate), który wprowadza losowe zmiany do osobnika za pomocą funkcji mutate_individual.

toolbox.register("select"…):

- Rejestruje metodę selekcji turniejowej (select), w której wybierani są najlepsi osobnicy spośród losowych grup o rozmiarze tournsize=3.

Wszystkie te funkcje i operatory są później wykorzystane w procesie ewolucji, tworzenia, oceny i modyfikowania populacji osobników.

```
population = toolbox.population(n=7000)    # Tworzenie populacji

max_generations = 200          # maksymalna liczba generacji
crossover_probability = 0.8    # prawdopodobieństwo skrzyżowania
mutation_probability = 0.2     # prawdopodobieństwo mutacji

elitism_size = 1            # liczba elitarnych osobników w generacji
stagnations = 20                  # licznik stagnacji
mutation_tempo = 0.02         # jak szybko ma się zwiększać szansa mutacji
max_mut_prob = 0.8            # jaka może być max wartość szansy mutacji
```

W tym fragmencie kodu przedstawione są praktycznie wszystkie parametry które możemy zmieniać aby wpłynąć na to jak działa algorytm ewolucyjny.

Poza podstawowymi takimi jak maksymalna liczba generacji, szansa mutacji czy rekombinacji, zastosowaliśmy dwa rozwiązania które poprawiają wydajność algorytmu.

Pierwsze z nich to elitarność która sprawia, że określona liczba najlepszych osobników danej generacji przechodzi do następnej bez zmian. Pozwala to na ochronę najlepszego rozwiązania przed mutacją.

Drugie rozwiązanie, to stopniowe zwiększanie szansy na wystąpienie mutacji w przypadku kiedy algorytm utknie w minimum lokalnym i wynik nie poprawia się od określonej liczby generacji. Implementacja w głównej pętli algorytmu.

```
# Ewaluacja początkowej populacji
for ind in population:
    ind.fitness.values = toolbox.evaluate(ind)

best_fitness_previous = 999    # Poprzednia najlepsza wartość fitness
start_mut_prob = mutation_probability
stagnation_counter = 0
```

Ten fragment kodu jest odpowiedzialny za ocenę populacji startowej oraz inicjalizację kilku zmiennych potrzebnych w głównej o pętli algorytmu. Zmienna **best_fitness_previous** jest ustawiana na wysoką początkową wartość, co umożliwia porównywanie wyników w kolejnych generacjach. **start_mut_prob** zapisuje początkowe prawdopodobieństwo mutacji, aby można je było do niego powrócić po wyjściu z minimum lokalnego, a **stagnation_counter** zostaje zainicjalizowany jako licznik stagnacji, który będzie śledził brak poprawy najlepszego wyniku fitness w kolejnych generacjach.

```
for gen in range(max_generations):

    # Tworzenie potomstwa przez krzyżowanie i mutacje
    offspring = algorithms.varAnd(population, toolbox, cxpb=crossover_probability,
mutpb=mutation_probability)

    # Ocena nowego potomstwa
    fits = toolbox.map(toolbox.evaluate, offspring)
    for fit, ind in zip(fits, offspring):
        ind.fitness.values = fit

    # Wybór elit z obecnej populacji
    elites = tools.selBest(population, elitism_size)

    # Selekcja pozostałych osobników
    selected = toolbox.select(offspring, k=len(population) - elitism_size)

    # Tworzenie nowej populacji z elit i wybranych osobników
    population = elites + selected

    # Zbieranie statystyk
    fits = [ind.fitness.values[0] for ind in population]
    best_fitness = min(fits)
    mean_fitness = sum(fits) / len(fits)
    print(f"Generacja {gen}: Najlepsze dopasowanie = {best_fitness}, Średnie
dopasowanie = {mean_fitness}")

    # Sprawdzenie, czy znaleziono rozwiązanie
    best_ind = tools.selBest(population, 1)[0]          # najelpszy osobnik generacji
    if best_ind.fitness.values[0] == 0:
        print("Znaleziono rozwiązanie w generacji", gen)
        break

    # Mechanizm adaptacyjny
    if best_fitness == best_fitness_previous:   # Jeśli wynik się powtórzył
        stagnation_counter += 1                 # zwiększ licznik
    else:
        if best_fitness < best_fitness_previous:            # Jeśli wynik się poprawił
            mutation_probability = start_mut_prob # reset szansy na mutację
        stagnation_counter = 0
        best_fitness_previous = best_fitness
```

```
    if stagnation_counter >= stagnations:
        mutation_probability = min(mutation_probability + mutation_tempo, max_mut_prob)
        print(f"Stagnacja od {stagnations} generacji. Zwiększam prawdopodobieństwo mutacji do {mutation_probability:.2f}")
        stagnation_counter = 0
```

Ten fragment kodu przedstawia główną pętlę algorytmu ewolucyjnego. Uruchamia się ona zdefiniowaną jako **max_generations** liczbę razy i każdy przebieg reprezentuje życie jednej generacji.

```
print("Najlepszy osobnik:")
print(best_ind)    # rozwiązana plansza
print("Fitness:", best_ind.fitness.values[0])
```

Na koniec wyświetla nam się najlepszy osobnik i jego fitness.

Przykładowe wyjście programu:

```
Generacja 0: Najlepsze dopasowanie = 18.0, Średnie dopasowanie = 31.753285714285713
Generacja 1: Najlepsze dopasowanie = 18.0, Średnie dopasowanie = 29.071714285714286
Generacja 2: Najlepsze dopasowanie = 17.0, Średnie dopasowanie = 26.757428571428573
Generacja 3: Najlepsze dopasowanie = 15.0, Średnie dopasowanie = 24.732285714285716
Generacja 4: Najlepsze dopasowanie = 14.0, Średnie dopasowanie = 22.76214285714286
Generacja 5: Najlepsze dopasowanie = 12.0, Średnie dopasowanie = 20.861285714285714
Generacja 6: Najlepsze dopasowanie = 10.0, Średnie dopasowanie = 19.065428571428573
Generacja 7: Najlepsze dopasowanie = 7.0, Średnie dopasowanie = 17.359142857142857
Generacja 8: Najlepsze dopasowanie = 4.0, Średnie dopasowanie = 15.790285714285714
Generacja 9: Najlepsze dopasowanie = 4.0, Średnie dopasowanie = 14.15242857142857
Generacja 10: Najlepsze dopasowanie = 4.0, Średnie dopasowanie = 12.560857142857143
Generacja 11: Najlepsze dopasowanie = 2.0, Średnie dopasowanie = 11.033714285714286
Generacja 12: Najlepsze dopasowanie = 0.0, Średnie dopasowanie = 9.505428571428572
Znaleziono rozwiązanie w generacji 12
Najlepszy osobnik:
[[5 3 4 6 7 8 9 1 2]
 [6 7 2 1 9 5 3 4 8]
 [1 9 8 3 4 2 5 6 7]
 [8 5 9 7 6 1 4 2 3]
 [4 2 6 8 5 3 7 9 1]
 [7 1 3 4 2 9 8 5 6]
 [9 6 1 5 3 7 2 8 4]
 [2 8 7 9 1 4 6 3 5]
 [3 4 5 2 8 6 1 7 9]]
Fitness: 0.0
```

4. Wyniki eksperymentów

Aby ocenić skuteczność zaimplementowanego algorytmu ewolucyjnego w rozwiązywaniu Sudoku, przeprowadziliśmy serię eksperymentów. Algorytm został uruchomiony **10 razy** dla planszy o poziomie trudności "łatwy", aby sprawdzić jego wydajność i stabilność.

4.1. Parametry algorytmu

We wszystkich eksperymentach zastosowano następujące parametry algorytmu:

- **Rozmiar populacji**: 7000
- **Maksymalna liczba generacji**: 100
- **Prawdopodobieństwo krzyżowania**: 0,7
- **Prawdopodobieństwo mutacji**: 0,3
- **Rozmiar turnieju przy selekcji**: 7
- **Elitarność**: 2 najlepsze osobniki przechodzą bez zmian do kolejnej generacji
- **Adaptacja mutacji**: Jeśli przez 5 generacji nie ma poprawy, zwiększamy prawdopodobieństwo mutacji o 0,05, maksymalnie do 0,8

Parametry te zostały dobrane eksperymentalnie, aby zapewnić równowagę między eksploracją nowych rozwiązań a eksploatacją już znalezionych.

4.2. Przebieg eksperymentów

Podczas każdego uruchomienia zbieraliśmy informacje o:

- **Najlepszym**, **najgorszym** i **średnim** fitness w każdej generacji
- **Liczbie generacji** potrzebnej do znalezienia rozwiązania (fitness = 0)

Wyniki z 10 uruchomień przedstawiają się następująco:

```
Uruchomienie 1/10
Znaleziono rozwiązanie w generacji 10
Uruchomienie 2/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 3/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 4/10
Znaleziono rozwiązanie w generacji 9
Uruchomienie 5/10
Znaleziono rozwiązanie w generacji 9
Uruchomienie 6/10
Znaleziono rozwiązanie w generacji 9
Uruchomienie 7/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 8/10
Znaleziono rozwiązanie w generacji 11
Uruchomienie 9/10
Znaleziono rozwiązanie w generacji 9
Uruchomienie 10/10
Znaleziono rozwiązanie w generacji 10
```

4.3. Analiza wyników

Liczba generacji do znalezienia rozwiązania

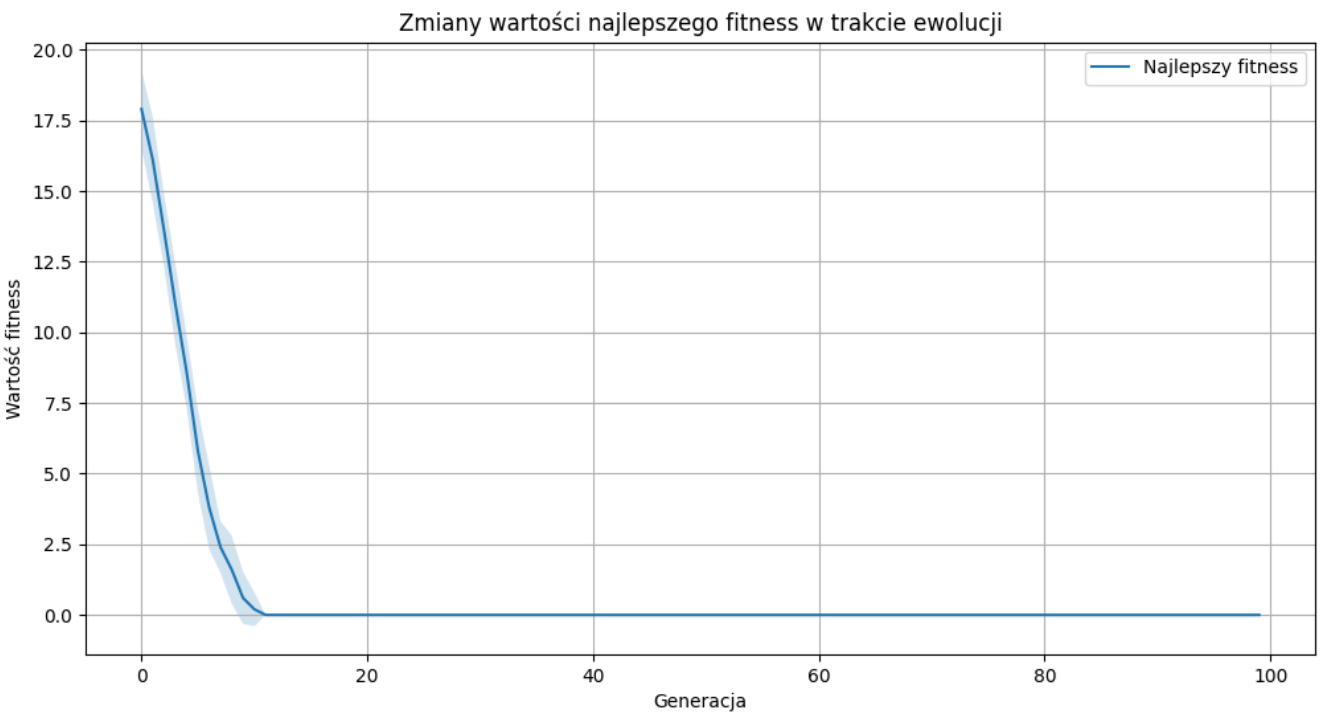
- **Najmniejsza liczba generacji**: 8
- **Największa liczba generacji**: 11
- **Średnia liczba generacji**: 9,1
- **Odchylenie standardowe liczby generacji**: około 0,94

Algorytm konsekwentnie znajdował poprawne rozwiązanie w niewielkiej liczbie generacji, co świadczy o jego skuteczności i stabilności.

Wykresy zmian fitness

Poniżej przedstawiamy wykresy ilustrujące zmiany wartości fitness w trakcie generacji.

Najlepszy fitness w generacjach



Opis wykresu:

- Wartość najlepszego fitness szybko maleje i zbliża się do zera, co oznacza, że algorytm znajduje coraz lepsze rozwiązania.
- Niskie odchylenie standardowe wskazuje na stabilność wyników między różnymi uruchomieniami.

Najgorszy fitness w generacjach

Opis wykresu:

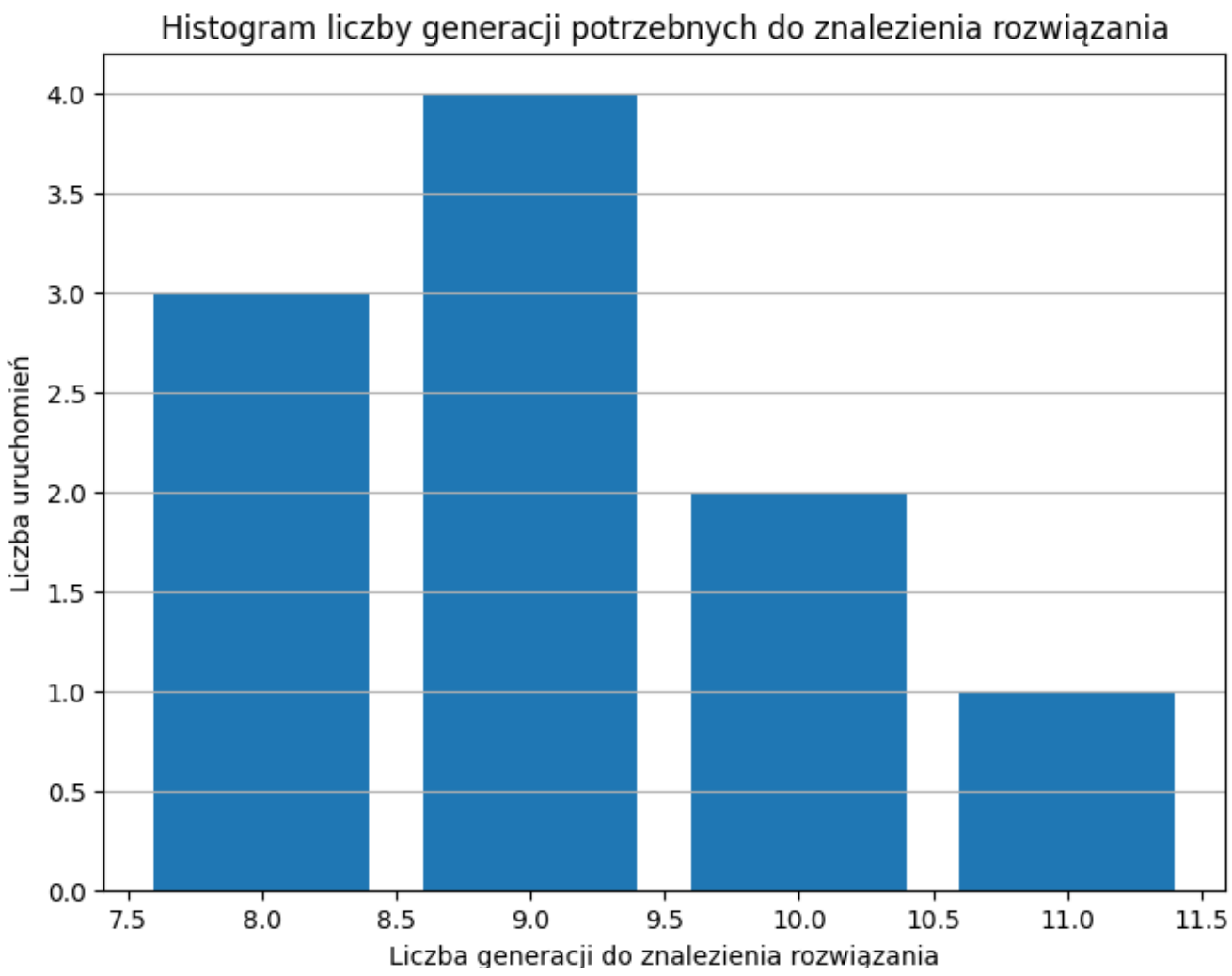
- Najgorszy fitness również maleje, ale wolniej niż najlepszy.
- Wyższe odchylenie standardowe na początku wskazuje na większą różnorodność populacji, która zmniejsza się w miarę postępu generacji.

Średni fitness w generacjach

Opis wykresu:

- Średni fitness populacji stopniowo maleje, co pokazuje ogólną poprawę jakości osobników.
- Spadek odchylenia standardowego wskazuje na konwergencję populacji.

Histogram liczby generacji do znalezienia rozwiązania



Opis wykresu:

- Większość uruchomień zakończyła się znalezieniem rozwiązania między 8 a 10 generacją.
- Histogram pokazuje skupienie wyników wokół średniej liczby generacji.

4.4. Najlepsze znalezione rozwiązanie

Poniżej przedstawiamy jedno z rozwiązań Sudoku znalezionych przez algorytm:

Najlepsze znalezione rozwiązanie:

```
5 3 4 | 6 7 8 | 9 1 2
6 7 2 | 1 9 5 | 3 4 8
1 9 8 | 4 2 3 | 5 6 7
---------------------
8 5 9 | 7 6 1 | 4 2 3
4 2 6 | 8 5 3 | 7 9 1
7 1 3 | 9 4 2 | 8 5 6
---------------------
9 6 1 | 5 3 7 | 2 8 4
2 8 7 | 4 1 9 | 6 3 5
3 4 5 | 2 8 6 | 1 7 9
```

Analiza rozwiązania:

- W każdym wierszu, kolumnie i bloku 3x3 występują liczby od 1 do 9 bez powtórzeń.
- Rozwiązanie jest zgodne z zasadami Sudoku i stanowi poprawne wypełnienie planszy.

4.5. Wnioski

- **Skuteczność algorytmu**: Algorytm ewolucyjny skutecznie rozwiązał problem Sudoku, znajdując poprawne rozwiązanie w każdym z 10 uruchomień.
- **Szybkość konwergencji**: Znalezienie rozwiązania w średnio 9,1 generacji świadczy o wysokiej efektywności algorytmu.
- **Stabilność wyników**: Niewielkie odchylenie standardowe liczby generacji do znalezienia rozwiązania wskazuje na stabilność algorytmu.
- **Adaptacja mutacji**: Mechanizm zwiększania prawdopodobieństwa mutacji w przypadku stagnacji pomógł uniknąć utknięcia w lokalnych minimach.

4.6. Porównanie wyników dla różnych poziomów trudności

Aby ocenić, jak poziom trudności planszy Sudoku wpływa na efektywność algorytmu ewolucyjnego, przeprowadziliśmy dodatkowe eksperymenty na planszach o poziomach trudności **"łatwy"** i **"średni"**. Celem było porównanie liczby generacji potrzebnych do znalezienia rozwiązania w zależności od złożoności problemu.

4.6.1. Eksperymenty z planszą o łatwym poziomie trudności

Wyniki eksperymentów dla poziomu "łatwy":

```
Uruchomienie 1/10
Znaleziono rozwiązanie w generacji 10
Uruchomienie 2/10
Znaleziono rozwiązanie w generacji 9
Uruchomienie 3/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 4/10
Znaleziono rozwiązanie w generacji 11
Uruchomienie 5/10
Znaleziono rozwiązanie w generacji 11
Uruchomienie 6/10
Znaleziono rozwiązanie w generacji 11
Uruchomienie 7/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 8/10
Znaleziono rozwiązanie w generacji 8
Uruchomienie 9/10
Znaleziono rozwiązanie w generacji 7
Uruchomienie 10/10
Znaleziono rozwiązanie w generacji 9

Średnia liczba generacji dla poziomu 'łatwy': 9.20 (odchylenie standardowe: 1.40)
```

4.6.2. Eksperymenty z planszą o średnim poziomie trudności

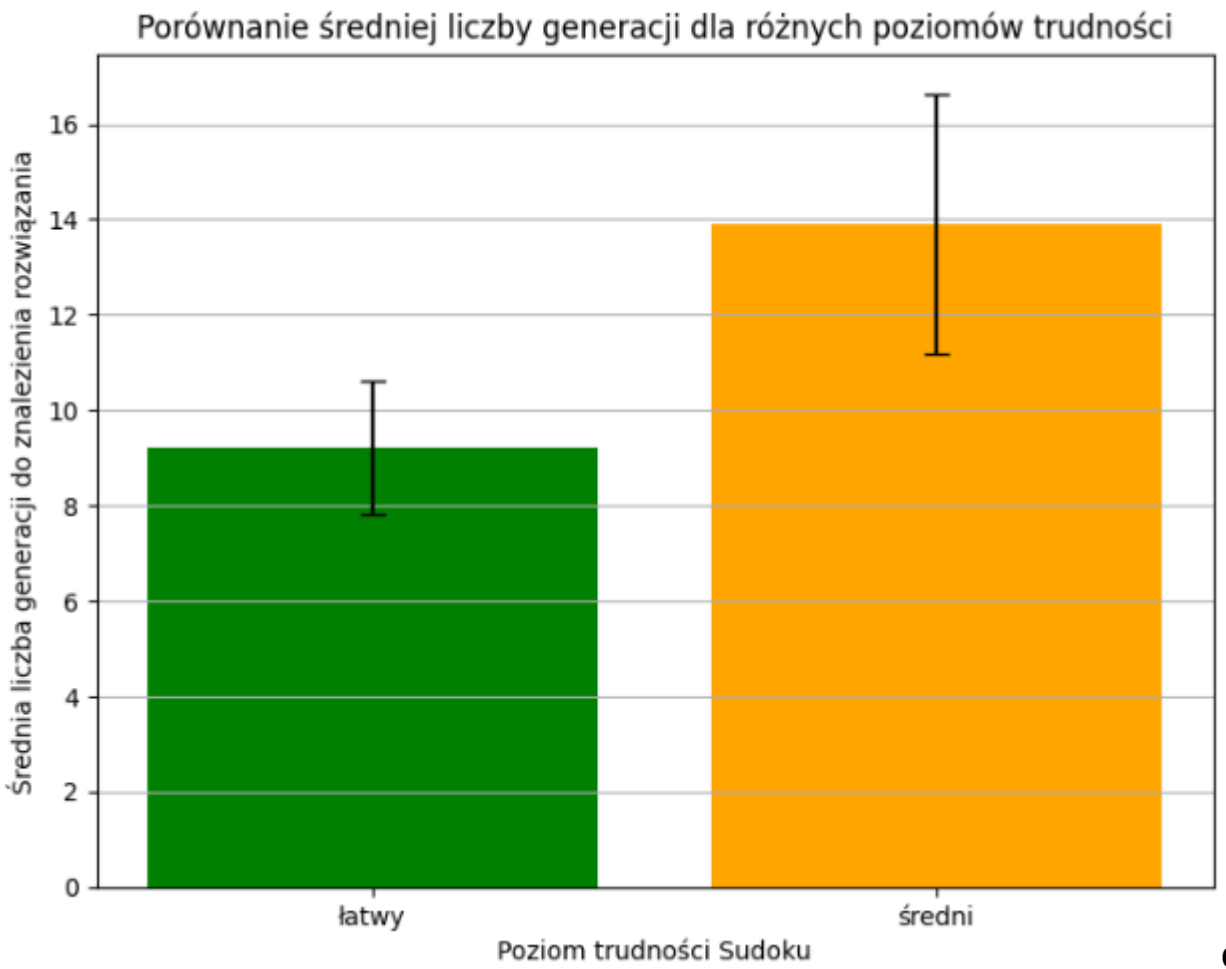
Wyniki eksperymentów dla poziomu "średni":

```
Uruchomienie 1/10
Znaleziono rozwiązanie w generacji 16
Uruchomienie 2/10
Znaleziono rozwiązanie w generacji 14
Uruchomienie 3/10
Znaleziono rozwiązanie w generacji 14
Uruchomienie 4/10
Znaleziono rozwiązanie w generacji 20
Uruchomienie 5/10
Znaleziono rozwiązanie w generacji 13
Uruchomienie 6/10
Znaleziono rozwiązanie w generacji 16
Uruchomienie 7/10
Znaleziono rozwiązanie w generacji 12
Uruchomienie 8/10
Znaleziono rozwiązanie w generacji 13
Uruchomienie 9/10
Znaleziono rozwiązanie w generacji 10
Uruchomienie 10/10
Znaleziono rozwiązanie w generacji 11

Średnia liczba generacji dla poziomu 'średni': 13.90 (odchylenie standardowe: 2.74)
```

4.6.3. Porównanie wyników

Poniżej przedstawiamy porównanie średniej liczby generacji potrzebnych do znalezienia rozwiązania dla poziomów trudności **"łatwy"** i **"średni"**:



Opis wykresu:

- **Poziom "łatwy"**: średnia liczba generacji to **9,20**.
- **Poziom "średni"**: średnia liczba generacji to **13,90**.
- Widać, że wraz ze wzrostem poziomu trudności planszy, algorytm potrzebuje więcej generacji, aby znaleźć rozwiązanie.

4.6.4. Wnioski z porównania

- **Wpływ trudności na efektywność**: Zwiększenie poziomu trudności planszy skutkuje wzrostem średniej liczby generacji potrzebnych do znalezienia rozwiązania.
- **Stabilność algorytmu**: Algorytm nadal skutecznie znajduje poprawne rozwiązania dla obu poziomów trudności, choć zajmuje to więcej czasu w przypadku trudniejszej planszy.
- **Skalowalność**: Algorytm wykazuje zdolność do skalowania się wraz ze wzrostem złożoności problemu, co świadczy o jego potencjale do rozwiązywania jeszcze trudniejszych zadań.

Rekomendacje:

- **Dostosowanie parametrów**: Dla trudniejszych plansz warto rozważyć dostosowanie parametrów algorytmu, takich jak zwiększenie maksymalnej liczby generacji czy rozmiaru populacji, aby utrzymać efektywność.
- **Ulepszenie operatorów genetycznych**: Można eksperymentować z różnymi metodami krzyżowania i mutacji, aby poprawić zdolność algorytmu do eksploracji przestrzeni rozwiązań w trudniejszych problemach.

5. Podsumowanie

W ramach projektu przeanalizowaliśmy i zaimplementowaliśmy algorytm ewolucyjny do rozwiązywania problemu Sudoku. Naszym celem było zbadanie skuteczności takiego podejścia oraz ocena jego wydajności w zależności od poziomu trudności planszy.

Rezultaty:

- **Skuteczność algorytmu:** Algorytm ewolucyjny okazał się skutecznym narzędziem do rozwiązywania Sudoku. W przeprowadzonych eksperymentach zawsze znajdował poprawne rozwiązanie, niezależnie od poziomu trudności planszy.
- **Wydajność i szybkość konwergencji:** Dla planszy o poziomie trudności "łatwy" algorytm znajdował rozwiązanie średnio w 9,20 generacji, natomiast dla planszy "średniej" potrzebował średnio 13,90 generacji. Wskazuje to na zwiększenie czasu potrzebnego do znalezienia rozwiązania wraz ze wzrostem złożoności problemu, jednak algorytm nadal zachowuje wysoką efektywność.
- **Stabilność wyników:** Niewielkie odchylenia standardowe liczby generacji do znalezienia rozwiązania dla obu poziomów trudności świadczą o stabilności algorytmu i jego niezawodności w różnych warunkach.
- **Adaptacja parametrów:** Wprowadzenie mechanizmu adaptacyjnego dla prawdopodobieństwa mutacji pozwoliło na uniknięcie stagnacji i utknięcia w lokalnych minimach, co pozytywnie wpłynęło na efektywność algorytmu.

Wnioski:

Algorytmy ewolucyjne mogą być efektywnie stosowane do rozwiązywania złożonych problemów kombinatorycznych, takich jak Sudoku. Ich zdolność do przeszukiwania dużych przestrzeni rozwiązań

oraz elastyczność w dostosowywaniu parametrów sprawiają, że są one wartościowym narzędziem w optymalizacji.

Możliwości dalszego rozwoju:

- **Dostosowanie parametrów dla trudniejszych problemów:** Aby zwiększyć efektywność algorytmu dla trudniejszych plansz, można eksperymentować z różnymi wartościami parametrów, takimi jak rozmiar populacji czy maksymalna liczba generacji.
- **Wprowadzenie zaawansowanych operatorów genetycznych:** Implementacja bardziej zaawansowanych metod krzyżowania i mutacji może poprawić zdolność algorytmu do eksploracji przestrzeni rozwiązań i przyspieszyć konwergencję.
- **Zastosowanie algorytmów wielokryterialnych:** Rozważenie wielu kryteriów oceny może prowadzić do znalezienia bardziej optymalnych rozwiązań w kontekście różnych metryk jakości.