WYDZIAŁ
MATEMATYKI
I FIZYKI STOSOWANEJ
POLITECHNIKI RZESZOWSKIEJ

IMPLEMENTACJA ALGORYTMU GENETYCZNEGO NA PRZYKŁADZIE

WPROWADZENIE DO PROGRAMOWANIA W JĘZYKU PYTHON

13.11.2023

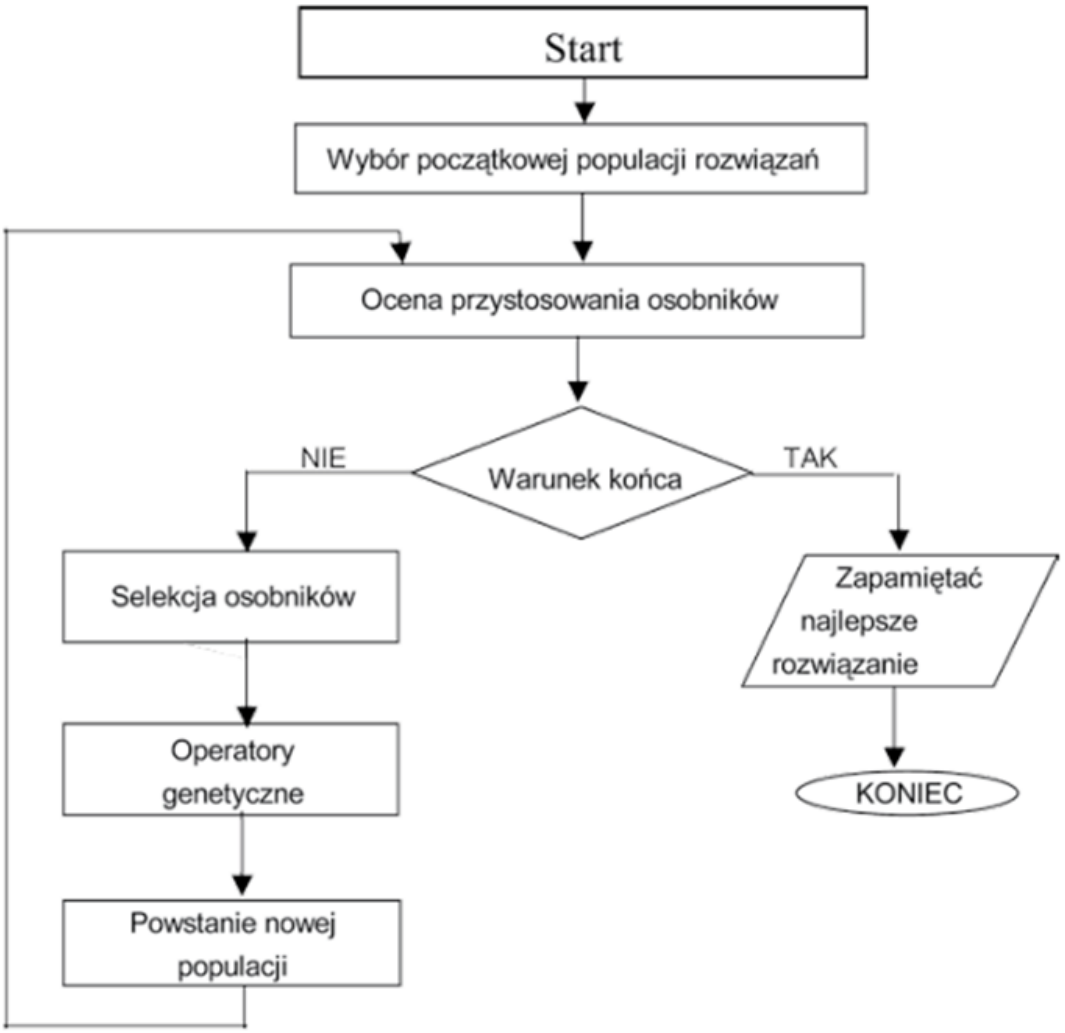
1. WPROWADZENIE

Algorytmy genetyczne (AG) to rodzaj heurystyki optymalizacyjnej opartej na mechanizmach dziedziczenia genetycznego i inspirowanej teorią ewolucji naturalnej Karola Darwina. Są one często stosowane do rozwiązywania problemów optymalizacyjnych, szczególnie w sytuacjach, gdzie tradycyjne metody matematyczne są niewystarczające. Algorytmy genetyczne opierają się na operatorach inspirowanych biologią, takich jak mutacja, krzyżowanie i selekcja .

2. Podstawowe pojęcia związane z algorytmem genetycznym

- Gen, jak wiadomo z genetyki, jest podstawową materialną jednostką dziedziczenia, która jest związana z przekazywaniem poszczególnych cech dziedziczenia organizmu. Gen to pojedynczy element genotypu, w szczególności chromosomu. Gen może być również nazwany cechą, znakiem bądź detektorem.
- Populacja - to zbiór potencjalnych osobników o określonej liczebności w danym pokoleniu algorytmu genetycznego
- Selekcja - to proces wybierania osobników z populacji do reprodukcji na podstawie ich funkcji przystosowania. Osobniki o lepszej funkcji przystosowania mają większą szansę być wybrane, co imituje zjawisko naturalnej selekcji
- Reprodukcja - to proces, w którym osobniki z wybranej populacji są używane do generowania potomstwa, które zastąpi starsze osobniki w kolejnym pokoleniu. W kontekście algorytmów genetycznych, proces reprodukcji obejmuje krzyżowanie i mutację.
- Krzyżowanie - to operator genetyczny, w którym dwa rodziców są używane do stworzenia potomstwa poprzez wymianę fragmentów ich genotypów.
- Mutacja to losowa zmiana jednego lub więcej genów w genotypie osobnika. Jest to mechanizm wprowadzania różnorodności genetycznej w populacji, co pozwala na eksplorację nowych obszarów przestrzeni rozwiązań.
- Genotyp, czyli struktura jest zespołem chromosomów danego osobnika. W genetyce genotyp oznacza skład osobnika, w którym może występować więcej niż jeden chromosom. Komórki człowieka zawierają 46 chromosomów, natomiast w algorytmach genetycznych z reguły przyjmuje się, że genotyp składa się z jednego chromosomu, który jest traktowany jako osobnik populacji.
- Chromosomy, czyli łańcuchy stanowią uporządkowane ciągi genów. Długość chromosomów będzie uzależniona od warunków zadania. Warto podkreślić, że w organizmach żywych długość chromosomów może wynosić nawet do tysięcy genów.

3. Kroki algorytmu genetycznego



1) Inicjalizacja populacji:

Na początku algorytmu genetycznego tworzona jest początkowa populacja, która składa się z pewnej liczby losowych osobników. Każdy osobnik w populacji reprezentuje potencjalne rozwiązanie problemu optymalizacyjnego i jest przedstawiany za pomocą genotypu.

2) Selekcja:

Proces selekcji polega na wyborze osobników do reprodukcji w oparciu o ich funkcję przystosowania. Funkcja przystosowania ocenia, jak dobre jest dane rozwiązanie w kontekście rozwiązywanego problemu. Osobniki o lepszych wartościach funkcji przystosowania mają większą szansę być wybrane.

3) Krzyżowanie:

Krzyżowanie jest operatorem genetycznym, który polega na kombinowaniu genotypów dwóch rodziców w celu stworzenia potomstwa. Punkt krzyżowania określa, gdzie następuje wymiana informacji genetycznej między rodzicami. Celem krzyżowania jest tworzenie potomstwa, które łączy korzystne cechy obu rodziców.

4) Mutacja:

Mutacja to losowe wprowadzenie zmian genetycznych w genotypie niektórych osobników w populacji. Operacja ta ma na celu wprowadzenie nowej różnorodności genetycznej, co pomaga w eksploracji przestrzeni rozwiązań. Mutacja może modyfikować pojedyncze geny lub fragmenty genotypu.

5) Ocena przystosowania:

Każdy osobnik w populacji jest oceniany pod kątem jego funkcji przystosowania. Funkcja przystosowania może być zdefiniowana w zależności od charakterystyki problemu optymalizacyjnego i mierzy, jak dobrze dany osobnik spełnia założone kryteria.

6) Zastąpienie populacji:

Po ocenie przystosowania aktualizowana jest populacja. Osobniki o lepszych wartościach przystosowania zastępują te o niższych wartościach. Proces ten pomaga utrzymać lub zwiększyć poziom jakości rozwiązań w populacji.

7) Warunki zakończenia:

Algorytm genetyczny powtarza te kroki przez określoną liczbę iteracji lub do momentu spełnienia pewnych warunków zakończenia. Warunki te mogą obejmować osiągnięcie określonej wartości funkcji przystosowania, przekroczenie maksymalnej liczby generacji, czy innych kryteriów zakończenia.

4. Zastosowania

Algorytmy genetyczne mogą być użyte do rozwiązywania problemów, w których inne metody są nieskuteczne i mało efektowne. Często są to problemy nieliniowe, nieciągłe, źle uwarunkowane lub trudne do matematycznego sformułowania. Algorytmy genetyczne stosuje się jako doskonałe narzędzie do poprawienia efektywności innych metod optymalizacji poprzez wskazanie dobrych punktów startowych dla tych metod. Dzięki swojej wydajności i prostocie implementacji AG znalazły szerokie zastosowanie w rozwiązywaniu problemów takich jak: szeregowanie zadań, konstrukcja strategii inwestycyjnych, modelowanie finansowe, optymalizacja funkcji, podejmowanie decyzji oraz minimalizacja kosztów czy harmonogramowanie pracy itp. W zarządzaniu produkcją, algorytmy

genetyczne zastosowano do znalezienia optymalnych wartości wielu parametrów, które reprezentują wagi pewnych kryteriów. Za pomocą algorytmu genetycznego prognozowano popyt na określone produkty w celu ustalenia wielkości produkcji w danym okresie planistycznym. W budowie maszyn wykorzystano algorytmy genetyczne do zaprojektowania odpowiednich cech sieci przemysłowej płynów. Często wykorzystuje się tę metodę do optymalizacji kształtu poprzez zminimalizowanie masy własnej z uwzględnieniem zachowania odpowiednich właściwości wytrzymałościowych. W niektórych pracach algorytmy użyto do optymalizacji parametrów technologicznych niektórych metod spajania materiałów czy optymalizacji kosztów całego procesu technologicznego. W eksploatacji maszyn algorytmy genetyczne znalazły zastosowanie do kontroli sposobu wytwarzania energii elektrycznej z zastosowaniem kilku jej źródeł jak również do rozwiązania odwrotnego zadania przepływu ciepła. Kolejnymi przykładami tej metody poszukiwań są optymalizacje sieci komputerowej, projektowanie laminatów czy poszukiwanie strategii rozwiązania pewnych problemów sterowania.

5. Zalety

- elastyczność - AG mogą być dostosowywane do różnorodnych problemów optymalizacyjnych, zarówno w dziedzinie ciągłej, jak i dyskretnej
- równoległość - łatwość w równoległym przetwarzaniu, co pozwala na jednoczesną ewolucję wielu osobników, co przyspiesza proces optymalizacji
- brak wymagań analitycznych - AG nie wymagają znajomości analitycznej funkcji przystosowania czy pochodnych, co jest korzystne w przypadku problemów, dla których trudno jest uzyskać analityczne rozwiązania
- skuteczność w przeszukiwaniu przestrzeni rozwiązań - dzięki kombinacji operacji krzyżowania, mutacji i selekcji, AG mogą efektywnie przeszukiwać przestrzeń rozwiązań w poszukiwaniu optymalnych lub zbliżonych do optymalnych rozwiązań
- znajdowanie rozwiązań globalnych - zdolność do znajdowania rozwiązań globalnych, nawet w przypadku funkcji przystosowania o nieliniowym i skomplikowanym charakterze
- szybkość - mogą generować dobre rozwiązania przybliżone w krótkim czasie, co jest szczególnie korzystne w przypadku problemów optymalizacyjnych trudnych do rozwiązania dokładnie

6. Ograniczenia

- brak gwarancji optymalności - AG nie zapewniają gwarancji znalezienia optymalnego rozwiązania, a jedynie dążą do znalezienia dobrego rozwiązania przybliżonego
- wybór parametrów - wybór odpowiednich parametrów, takich jak rozmiar populacji, prawdopodobieństwo krzyżowania i mutacji, może być trudny i czasochłonny
- zależność od struktury populacji początkowej - wyniki AG mogą być wrażliwe na strukturę początkowej populacji, co oznacza, że różne próby z tym samym problemem mogą prowadzić do różnych rezultatów
- wrażliwość na kodowanie - skuteczność AG może zależeć od wyboru odpowiedniego kodowania genotypu, co może być problematyczne w pewnych przypadkach
- wymagania obliczeniowe - AG mogą być czasochłonne, zwłaszcza przy rozwiązywaniu skomplikowanych problemów, co może wpływać na wydajność w niektórych zastosowaniach

7. Implementacja algorytmu

0. Używane biblioteki

```
# Wczytywanie używanej biblioteki
from numpy.random import randint, rand
```

1. Tworzenie Początkowej Populacji

Pierwszy blok kodu służy do wygenerowania początkowej populacji losowych ciągów bitowych. Wartości logiczne "True" i "False", stringi '0' i '1', lub wartości liczbowe 0 i 1 mogą reprezentować bity. W tym przypadku używamy wartości liczbowych, tworząc listę pop, gdzie każdy element to losowo generowany ciąg bitów o długości n_bits.

```
# Przykładowy kod tworzący populacje losowych ciągów bitowych
n_bits= 5
n_pop = 2
pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
print(pop)
```

```
[[1, 1, 0, 1, 0], [1, 1, 1, 1, 1]]
```

2. Pętla Wyliczenia Stałej Liczby Iteracji

Pętla for gen in range(n_iter) wyznacza stałą liczbę iteracji algorytmu genetycznego. Każda iteracja odpowiada jednemu pokoleniu w ewolucji.

```
for gen in range(n_iter):
...
```

3. Ocena Przystosowania

W każdej iteracji oceniamy przystosowanie każdego osobnika w populacji za pomocą funkcji celu (cel). Wyniki wartości funkcji, którą minimalizujemy, są przechowywane w liście wyniki.

```
wyniki = [cel(c) for c in pop]
```

4. Wybór Rodziców

Następnie wybieramy rodziców do reprodukcji za pomocą procedury selekcji. Funkcja selekcja wybiera jednego rodzica z populacji na podstawie ich funkcji przystosowania. Wartość k jest ustalona na 3 z domyślnym argumentem. Wywołujemy funkcję selekcja dla każdej pozycji w populacji, aby utworzyć listę rodziców.

```
def selekcja(pop, wyniki, k=3):
    # pierwszy losowy wybór
    selekcja_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # sprawdzanie czy wybrany został lepszy
        if wyniki[ix] < wyniki[selekcja_ix]:
            selekcja_ix = ix
    return pop[selekcja_ix]

wybrani = [selekcja(pop, wyniki) for _ in range(n_pop)]
```

5. Krzyżowanie i Mutacja

Po wyborze rodziców następuje krzyżowanie i mutacja. Funkcja krzyzowanie tworzy potomków poprzez kombinację genotypów rodziców. Ta funkcja będzie uwzględniać dwoje rodziców i współczynnik krzyżowania. Współczynnik krzyżowania to hiperparametr określający, czy krzyżowanie jest wykonywane, czy nie, a jeśli nie, elementy nadrzędne są kopiowane do następnej generacji. Ten parametr to współczynnik prawdopodobieństwa i zazwyczaj ma dużą wartość bliską 1,0.

Poniższa funkcja krzyzowanie() implementuje krzyżowanie poprzez losowanie liczby z zakresu [0,1] w celu ustalenia, czy krzyżowanie jest wykonywane, a następnie wybiera prawidłowy punkt podziału, jeśli ma zostać przeprowadzone krzyżowanie.

```
# krzyżowanie dwóch rodziców w celu wygenerowania dwojga potomków
def krzyzowanie(p1, p2, r_cross):
    # potomkowie domyślnie są kopią rodziców
    c1, c2 = p1.copy(), p2.copy()
    # sprawdzanie w celu rekombinacji
    if rand() < r_cross:
          # wybór punktu krzyżowania, który nie znajduje się na końcu
ciągu
        pt = randint(1, len(p1)-2)
        # wykonanie krzyżowania
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]
```

Potrzebujemy również funkcji do przeprowadzenia mutacji. Ta procedura po prostu odwraca bity z niskim prawdopodobieństwem kontrolowanym przez hiperparametr „r_mut”.

```
# operator mutacji
def mutacja(bitstring, r_mut):
    for i in range(len(bitstring)):
        # sprawdzanie w celu przeprowadzenie mutacji
        if rand() < r_mut:
            # odwrócenie bitu
            bitstring[i] = 1 - bitstring[i]
```

6. Stworzenie Nowej Generacji

Następnie tworzymy nową generację, łącząc krzyżowanie i mutację dla każdej pary rodziców.

```
# stworzenie następnej generacji
dzieci = list()
for i in range(0, n_pop, 2):
    # połączenie wybranych rodziców w pary
    p1, p2 = wybrani[i], wybrani[i+1]
    # krzyżowanie i mutacja
    for c in krzyzowanie(p1, p2, r_cross):
        # mutacja
        mutacja(c, r_mut)
        # zapisz dla następnej generacji
        dzieci.append(c)
```

7. Algorytm Genetyczny

Ostateczna funkcja algorytm_genetyczny integruje wszystkie powyższe kroki, śledzi najlepsze rozwiązanie i zwraca je.

```
# algorytm genetyczny
def algorytm_genetyczny(cel, n_bits, n_iter, n_pop, r_krzyz, r_mut):
    # początkowa populacja losowego ciągu bitowego
    pop = [randint(0, 2, n_bits).tolist() for _ in range(n_pop)]
    # śledzenie najlepszego rozwiązania
    best, best_eval = 0, cel(pop[0])
    # wyliczanie pokoleń
    for gen in range(n_iter):
        # ocena wszystkich kandydatów w populacji
        wyniki = [cel(c) for c in pop]
        # sprawdzenie nowych najlepszych rozwiązań
        for i in range(n_pop):
            if wyniki[i] < best_eval:
                best, best_eval = pop[i], wyniki[i]
                  print(">%d, nowe najlepsze rozwiązanie f(%s) = %.3f" %
(gen,  pop[i], wyniki[i]))
        # wybór rodziców
        wybrani = [selekcja(pop, wyniki) for _ in range(n_pop)]
        # stworzenie nowej generacji
        dzieci = list()
        for i in range(0, n_pop, 2):
            # połączenie wybranych rodziców w pary
            p1, p2 = wybrani[i], wybrani[i+1]
            # krzyżowanie i mutacja
            for c in krzyzowanie(p1, p2, r_krzyz):
                mutacja(c, r_mut)
                # przechowywanie wyników dla następnego pokolenia
                dzieci.append(c)
        # zastąpienie populacji
        pop = dzieci
    return [best, best_eval]
```

8. OneMax problem

OneMax to jedno z klasycznych zadań optymalizacyjnych, które często jest używane do demonstracji działania algorytmów genetycznych. Problem polega na znalezieniu ciągu bitowego o maksymalnej długości, gdzie celem jest maksymalizacja sumy wartości bitów równych 1. Innymi słowy, dążymy do znalezienia ciągu, w którym wszystkie bity mają wartość 1.

Funkcja celu onemax przyjmuje ciąg bitowy x i zwraca negatywną sumę jego elementów. W tym przypadku, naszym celem jest minimalizacja tej wartości, co skutkuje maksymalizacją sumy wartości bitów równych 1.

```
# funkcja celu
def onemax(x):
    return -sum(x)
```

Następnie możemy skonfigurować wyszukiwanie. Wyszukiwanie będzie trwało 100 iteracji, a w naszych kandydujących rozwiązaniach użyjemy 20 bitów, co oznacza, że optymalna zgodność wyniesie -20,0. Wielkość populacji wyniesie 100, a my zastosujemy współczynnik krzyżowania wynoszący 90 procent i współczynnik mutacji wynoszący 5 procent. Ta konfiguracja została wybrana metodą prób i błędów.

```
# ilość iteracji
n_iter = 100
# ilość bitów
n_bits = 20
# wielkość populacji
n_pop = 100
# współczynnik krzyżowania
r_cross = 0.9
# współczynnik mutacji
r_mut = 1.0 / float(n_bits)
```

Poniższy fragment kodu uruchamia algorytm genetyczny na funkcji OneMax, używając wcześniej zdefiniowanych parametrów. Funkcja algorytm_genetyczny zwraca najlepsze znalezione rozwiązanie (best) i wartość funkcji przystosowania dla tego rozwiązania (score). Ostatecznie, program drukuje znalezione najlepsze rozwiązanie i wartość funkcji przystosowania.

```
# przeprowadzenie wyszukiwania algorytmu genetycznego

best, score = algorytm_genetyczny(onemax, n_bits, n_iter, n_pop,
r_cross, r_mut)

print('Znaleziono najlepsze rozwiązanie!')
```

```
print('f(%s) = %f' % (best, score))
```

```
>0, nowe najlepsze rozwiązanie f([0, 1, 1, 0, 0, 1, 0, 1, 1, 1, 0, 1,
0, 1, 1, 1, 0, 1, 0, 1]) = -12.000

>0, nowe najlepsze rozwiązanie f([1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1,
0, 0, 0, 1, 1, 1, 0, 1]) = -13.000

>0, nowe najlepsze rozwiązanie f([0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 0, 1, 1, 1, 1, 0, 0]) = -16.000

>1, nowe najlepsze rozwiązanie f([1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1,
1, 1, 1, 1, 1, 1, 1, 1]) = -18.000

>5, nowe najlepsze rozwiązanie f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 0, 1]) = -19.000

>6, nowe najlepsze rozwiązanie f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1]) = -20.000

Znaleziono najlepsze rozwiązanie!

f([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]) =
-20.000000
```

9. Implementacja algorytmu genetycznego (AG) do optymalizacji funkcji ciągłej

Celem algorytmu genetycznego będzie znalezienie takich wartości zmiennych x, dla których funkcja celu przyjmuje możliwie najniższą wartość, co odpowiada minimalizacji tej funkcji. Funkcja celu ma wartosci optymalne przy f(0, 0) = 0,0.

```python
# funkcja celu
def cel(x):
    return x[0]**2.0 + x[1]**2.0
```

Chcąc przeprowadzić optymalizację za pomocą algorytmu genetycznego, najpierw musimy zdefiniować granice każdej zmiennej wejściowej. Ten fragment kodu definiuje zakresy dla

zmiennych decyzyjnych, czyli przedziały, w których algorytm będzie przeszukiwał wartości zmiennych. W tym konkretnym przypadku granice są ustawione dla dwóch zmiennych decyzyjnych, z zakresem od -5 do 5.

```
# zakres
granice = [[-5.0, 5.0], [-5.0, 5.0]]
```

Przyjmiemy n_bits jako liczbę bitów na jedną zmienną wejściową do funkcji celu i ustawimy ją na 16 bitów. Zwiększenie liczby bitów może poprawić zdolność algorytmu genetycznego do dokładniejszego przeszukiwania przestrzeni poszukiwań, ale może też zwiększyć złożoność obliczeniową.

```
# liczba bitów na zmienną
n_bits = 16
```

Oznacza to, że nasz rzeczywisty ciąg bitów będzie miał (16 * 2) = 32 bity, biorąc pod uwagę dwie zmienne wejściowe.

Musimy odpowiednio zaktualizować nasz współczynnik mutacji, czyli prawdopodobieństwo, że pojedynczy bit w genotypie zostanie odwrócony (zmutowany).

```
# współczynnik mutacji
r_mut = 1.0 / (float(n_bits) * len(granice))
```

Następnie musimy się upewnić, że początkowa populacja tworzy losowe ciągi bitów, które są wystarczająco duże. Ta początkowa populacja genotypów jest wykorzystywana jako punkt wyjścia dla algorytmu genetycznego. W miarę kolejnych iteracji algorytmu, genotypy w populacji będą ewoluować poprzez operacje selekcji, krzyżowania i mutacji w kierunku lepszych rozwiązań.

```
# początkowa populacja losowego ciągu bitowego
pop = [randint(0, 2, n_bits*len(granice)).tolist() for _ in range(n_pop)]
print(pop)
```

Na koniec musimy rozszyfrować ciągi bitów na liczby przed oceną każdego z nich za pomocą funkcji celu.

Możemy to osiągnąć, najpierw rozszyfrowując każdy podciąg do liczby całkowitej, a następnie skalując liczbę całkowitą do żądanego zakresu. Otrzymamy wektor wartości w zakresie, który można następnie dostarczyć do funkcji celu, by umożliwić jego ocenę.

Poniższa funkcja rozszyfruj() przyjmuje granice funkcji, liczbę bitów na zmienną i ciąg bitów jako dane wejściowe i zwraca listę rozszyfrowanych wartości rzeczywistych.

```
# rozszyfruj ciąg bitów na liczby
def rozszyfruj(granice, n_bits, bitstring):
    rozszyfrowane = list()
    najwieksze = 2**n_bits
    for i in range(len(granice)):
        # wyodrębnij podciąg
        poczatek, koniec = i * n_bits, (i * n_bits)+n_bits
        podciag = bitstring[poczatek:koniec]
        # przekształć ciąg bitów na ciąg znaków
        znaki = ''.join([str(s) for s in podciag])
        # przekształć ciąg znaków na liczbę całkowitą
        calkowita = int(znaki, 2)
        # skaluj liczbę całkowitą do pożądanego zakresu
        wartosc = granice[i][0] + (calkowita/najwieksze) * (granice[i][1] - granice[i][0])
        # zapisz
        rozszyfrowane.append(wartosc)
    return rozszyfrowane
```

Łącząc to razem, pełny przykład algorytmu genetycznego do optymalizacji funkcji ciągłej znajduje się poniżej.

```
# importowanie funkcji randint i rand z biblioteki numpy

from numpy.random import randint
from numpy.random import rand

# funkcja celu
def cel(x):
    return x[0]**2.0 + x[1]**2.0

# dekodowanie ciągu bitów na liczby
def rozszyfruj(granice, n_bits, bitstring):
    rozszyfrowane = list()
    najwieksze = 2**n_bits
    for i in range(len(granice)):
        # wyodrębnij podciąg
        poczatek, koniec = i * n_bits, (i * n_bits)+n_bits
        podciag = bitstring[poczatek:koniec]
        # przekształć ciąg bitów na ciąg znaków
        znaki = ''.join([str(s) for s in podciag])
        # przekształć ciąg znaków na liczbę całkowitą
        calkowita = int(znaki, 2)
        # skaluj liczbę całkowitą do pożądanego zakresu
        wartosc = granice[i][0] + (calkowita/najwieksze) * (granice[i][1] - granice[i][0])
        # zapisz
        rozszyfrowane.append(wartosc)
    return rozszyfrowane
```

```python
# selekcja turniejowa
def selekcja(pop, wyniki, k=3):
    # pierwszy losowy wybór
    selekcja_ix = randint(len(pop))
    for ix in randint(0, len(pop), k-1):
        # sprawdź, czy lepsze
        if wyniki[ix] < wyniki[selekcja_ix]:
            selekcja_ix = ix
    return pop[selekcja_ix]

# krzyżowanie dwóch rodziców w celu wygenerowania dwojga potomków
def krzyzowanie(p1, p2, r_cross):
    # potomkowie domyślnie są kopią rodziców
    c1, c2 = p1.copy(), p2.copy()
    # sprawdzanie w celu rekombinacji
    if rand() < r_cross:
        # wybór punktu krzyżowania, który nie znajduje się na końcu ciągu
        pt = randint(1, len(p1)-2)
        # wykonanie krzyżowania
        c1 = p1[:pt] + p2[pt:]
        c2 = p2[:pt] + p1[pt:]
    return [c1, c2]
```

```python
# operator mutacji
def mutacja(bitstring, r_mut):
    for i in range(len(bitstring)):
        # sprawdzanie w celu przeprowadzenie mutacji
        if rand() < r_mut:
            # odwrócenie bitu
            bitstring[i] = 1 - bitstring[i]

# algorytm genetyczny
def algorytm_genetyczny(cel, granice, n_bits, n_iter, n_pop, r_cross, r_mut):
    # początkowa populacja losowego ciągu bitowego
    pop = [randint(0, 2, n_bits*len(granice)).tolist() for _ in range(n_pop)]
    # śledzenie najlepszego rozwiązania
    best, best_eval = 0, cel(rozszyfruj(granice, n_bits, pop[0]))
    # wyliczanie pokoleń
    for gen in range(n_iter):
        # rozszyfrowanie populacji
        rozszyfrowane = [rozszyfruj(granice, n_bits, p) for p in pop]
        # ocena wszystkich kandydatów w populacji
        wyniki = [cel(d) for d in rozszyfrowane]
        # sprawdzenie nowych najlepszych rozwiązań
        for i in range(n_pop):
            if wyniki[i] < best_eval:
                best, best_eval = pop[i], wyniki[i]
                print(">%d, nowe najlepsze rozwiązanie f(%s) = %f" % (gen,  rozszyfrowane[i], wyniki[i]))
        # wybór rodziców
        wybrani = [selekcja(pop, wyniki) for _ in range(n_pop)]
```

```
        # stworzenie nowej generacji
        dzieci = list()
        for i in range(0, n_pop, 2):
            # połączenie wybranych rodziców w pary
            p1, p2 = wybrani[i], wybrani[i+1]
            # krzyżowanie i mutacja
            for c in krzyzowanie(p1, p2, r_cross):
                # mutacja
                mutacja(c, r_mut)
                # przechowywanie wyników dla następnego pokolenia
                dzieci.append(c)
        # zastąpienie populacji
        pop = dzieci
    return [best, best_eval]

# zakres
granice = [[-5.0, 5.0], [-5.0, 5.0]]
# ilość iteracji
n_iter = 100
# liczba bitów na zmienną
n_bits = 16
# wielkość populacji
n_pop = 100
# współczynnik krzyżowania
r_cross = 0.9
# współczynnik mutacji
r_mut = 1.0 / (float(n_bits) * len(granice))
```

```
# przeprowadzenie wyszukiwania algorytmu genetycznego
best, score = algorytm_genetyczny(cel, granice, n_bits, n_iter, n_pop, r_cross, r_mut)
print('Koniec!')
rozszyfrowane = rozszyfruj(granice, n_bits, best)
print('f(%s) = %f' % (rozszyfrowane, score))
```

Executed at 2023.12.09 18:57:57 in 296ms

```
>0, nowe najlepsze rozwiązanie f([-0.169219970703125, 2.966003417968750]) = 8.825812
>0, nowe najlepsze rozwiązanie f([1.593170166015625, 2.271575927734375]) = 7.698248
>0, nowe najlepsze rozwiązanie f([-2.230224609375, 1.439666748046875]) = 7.046542
>0, nowe najlepsze rozwiązanie f([2.513427734375, -0.058288574218750]) = 6.320717
>0, nowe najlepsze rozwiązanie f([-1.000671386718750, -0.122528076171875]) = 1.016356
>0, nowe najlepsze rozwiązanie f([0.098876953125, -0.003662109375]) = 0.009790
>3, nowe najlepsze rozwiązanie f([0.020751953125, -0.005340576171875]) = 0.000459
>4, nowe najlepsze rozwiązanie f([-0.004119873046875, -0.012969970703125]) = 0.000185
>7, nowe najlepsze rozwiązanie f([-0.006866455078125, -0.008392333984375]) = 0.000118
>8, nowe najlepsze rozwiązanie f([-0.002136230468750, -0.007324218750]) = 0.000058
>12, nowe najlepsze rozwiązanie f([0.002136230468750, -0.005340576171875]) = 0.000033
>13, nowe najlepsze rozwiązanie f([0.002136230468750, -0.005187988281250]) = 0.000031
>14, nowe najlepsze rozwiązanie f([0.001525878906250, -0.000152587890625]) = 0.000002
>19, nowe najlepsze rozwiązanie f([0.000457763671875, -0.001373291015625]) = 0.000002
>23, nowe najlepsze rozwiązanie f([0.000152587890625, -0.001373291015625]) = 0.000002
>24, nowe najlepsze rozwiązanie f([0.000152587890625, -0.000610351562500]) = 0.000000
>28, nowe najlepsze rozwiązanie f([0.0, -0.000457763671875]) = 0.000000
>42, nowe najlepsze rozwiązanie f([0.000152587890625, -0.000152587890625]) = 0.000000
>45, nowe najlepsze rozwiązanie f([0.0, -0.000152587890625]) = 0.000000
Koniec!
f([0.0, -0.000152587890625]) = 0.000000
```

10. Bibliografia

https://machinelearningmastery.com/simple-genetic-algorithm-from-scratch-in-python/?fbclid=IwAR2vpGpMxPYv45yMnq5PcyTuxhlIYr1PlWfzuiKpodw3gatW5CR3cejCjZc

https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_introduction.htm

https://depot.ceon.pl/bitstream/handle/123456789/3287/licencjat_informatyka_2007.pdf

https://towardsdatascience.com/introduction-to-genetic-algorithms-including-example-code-e396e98d8bf3

https://en.wikipedia.org/wiki/Genetic_algorithm