Paweł Jędrzejczyk 224819

Podstawy sztucznej inteligencji
Laboratorium 2
Implementacja prostego algorytmu genetycznego

Definicja problemu

Dana jest funkcja f(x) = 2(x^2 + 1), x ∈<1…127>. Znajdź maksimum funkcji dla podanej dziedziny, używając do tego samodzielnie zaimplementowanego algorytmu genetycznego. Osobnikami algorytmu są wartości argumentu (jako genotypu używamy kodowania binarnego), natomiast funkcją przystosowania jest opisana wyżej funkcja kwadratowa. Zaimplementuj operacje krzyżowania i mutacji wraz z odpowiednimi prawdopodobieństwami. Dobierz odpowiednią ilość pokoleń jako warunek stopu. Jako metodę selekcji, zaimplementuj koło ruletki, gdzie prawdopodobieństwo selekcji każdego osobnika jest proporcjonalne do wartości jego funkcji oceny.

Rozwiązanie i opis

W celu wykonania zadania laboratoryjnego zaimplementowane zostało kilka funkcji odpowiadających za poszczególne wymagania realizacji rozwiązania problemu oraz pełniących funkcje pomocnicze dla poprawienia przejrzystości kodu.

Generowanie populacji

```
def generate_full_population():
    ret = []
    for x in range(0, 128):
        ret.append(x)
    return ret


def generate_random_population():
    ret = []
    for x in range(0, 128):
        ret.append(random.randint(0, 127))
    return ret
```

Zaimplementowane zostały dwie funkcje. Pierwsza generuje populację zawierającą wszystkie elementy dziedziny, druga zaś generuje populację losowych (ze zwracaniem) elementów dziedziny w liczności odpowiadającej liczbie elementów dziedziny. Obie funkcje generujące zwracają populację w liście.

Konwersja zawartości populacji

```
def cast_to_bin(population):
    ret = []
    for element in population:
        ret.append(bin(element).replace("0b", "").zfill(8))
    return ret

def cast_to_dec(population):
    ret = []
    for each in population:
```

```
        ret.append(int(each, 2))
    return ret
```

Zaimplementowane zostały dwie funkcje służące do konwersji zawartości populacji pomiędzy systemami binarnym i dziesiętnym. Funkcje przyjmują i zwracają listę.

Krzyżowanie

```
def cross(population):
    for i in range(0, int(len(population) / 2)):
        if random.random() < cross_probability:
            global crossings
            crossings += 2
            parent1 = population[2 * i]
            parent2 = population[2 * i + 1]
            child1 = parent1[:int(len(parent1)/2)] + parent2[int(len(parent2)/2):]
            child2 = parent2[:int(len(parent2)/2)] + parent1[int(len(parent1)/2):]
            population[2 * i] = child1
            population[2 * i + 1] = child2
```

Funkcja krzyżująca dokonuje krzyżowania pomiędzy osobnikami z pary poprzez podzielenie genotypu rodziców na 2 różne części i zestawienie części z różnych rodziców w dwa nowe osobniki. Prawdopodobieństwo krzyżowanie zależne jest od parametru ***cross_probability***
Funkcja krzyżująca przyjmuje populację jako listę, nie zwraca natomiast nic. Krzyżowanie następuje w miejscu.

Mutowanie

```
def swap_value(thing, sign):
    if thing[sign] == 0:
        thing = thing[:sign] + "1" + thing[sign + 1:]
    else:
        thing = thing[:sign] + "0" + thing[sign + 1:]
    return thing


def mutate(population):
    for each in population:
        if (random.random() < mutation_probability):
            global mutations
            mutations += 1
            each = swap_value(each, random.randint(1, len(each) - 1))
```

Funkcja mutująca działa poprzez zanegowanie jednego, losowego bitu w mutowanym osobniku. Mutowanie zależne jest od parametru ***mutation_probability***.
Prawdopodobieństwo wystąpienia mutacji sprawdzane jest dla każdego osobnika osobno.
Funkcja mutująca przyjmuje populację jako listę, nie zwraca natomiast nic. Mutowanie następuje w miejscu.
Dodatkowo zaimplementowana została funkcja ***swap_value*** realizująca negowanie bitu.

Koło ruletki

```
def get_new_generation(population):
    roulette_set = []
    new_population = []
    for each in population:
        for i in range(0, int(each, 2)):
            roulette_set.append(each)
    r_size = len(roulette_set)
    for j in range(0, len(population)):
```

```
        new_population.append(roulette_set[random.randint(0, r_size - 1)])
    return new_population
```

Funkcja realizująca koło ruletki działa w dwóch etapach. W pierwszym etapie generuje roboczą listę, która wypełniana jest osobnikami populacji w liczności odpowiadającej ich wartości w systemie dziesiętnym. W ten sposób realizowane jest założenie: im lepszy osobnik tym większe prawdopodobieństwo przejścia selekcji.
Następnie w drugim etapie z utworzonej listy roboczej wybierane są osobniki w liczbie odpowiadającej rozmiarowi populacji.
Funkcja ***get_new_generation*** przyjmuje i zwraca listę (starą i nową populację).

Algorytm

```
cross_probability = 0.71
mutation_probability = 0.08
crossings = 0
mutations = 0
number_of_generations = 150

# base_set = generate_full_population()
base_set = generate_random_population()
base_set_binary = cast_to_bin(base_set)

for i in range(0,number_of_generations):
    if i > 0:
        base_set_binary = get_new_generation(base_set_binary)
    random.shuffle(base_set_binary)
    mutate(base_set_binary)
    cross(base_set_binary)
    # print("generation " + str(i+1) + str(base_set_binary))
base_set_binary.sort()
print("Populacja wejściowa\n" + str(base_set))
print("Populacja po {0} generacjach\n".format(number_of_generations) +
str(cast_to_dec(base_set_binary)))
print("mutations: " + str(mutations))
print("crossings: " + str(crossings))
```

Algorytm dla każdej generacji (**number_of_generation**) dokonuje selekcji osobników z poprzedniej generacji (poza pierwszą), oraz dokonuje przemieszania osobników w populacji dla zwiększenia różnorodności krzyżowania. W kolejnych krokach następuje mutowanie i krzyżowanie. Po przejściu przez określoną na początku liczbę generacji następuje wydruk populacji wejściowej oraz końcowej, a także liczba osobników zmutowanych oraz utworzonych w wyniku krzyżowania

Przykładowy output:

```
Populacja wejściowa
[78, 92, 99, 112, 117, 35, 11, 91, 0, 75, 19, 91, 18, 110, 37, 69, 94, 82, 78, 35,
96, 119, 113, 104, 42, 6, 112, 111, 95, 72, 20, 100, 120, 3, 1, 113, 47, 91, 78,
64, 68, 40, 4, 47, 45, 61, 94, 119, 107, 44, 123, 25, 69, 10, 22, 50, 69, 96, 26,
42, 77, 21, 85, 38, 112, 79, 55, 44, 11, 51, 23, 21, 113, 125, 96, 33, 17, 83,
116, 55, 69, 46, 29, 29, 76, 84, 120, 88, 44, 53, 34, 117, 49, 89, 92, 72, 38, 68,
35, 76, 73, 48, 47, 100, 18, 25, 123, 98, 51, 108, 85, 53, 70, 20, 124, 61, 118,
12, 43, 98, 0, 106, 1, 28, 68, 88, 95, 59]
Populacja po 150 generacjach
[127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
```

```
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127,
127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127, 127]
mutations: 1541
crossings: 13526
```

Wnioski i obserwacje

Algorytm genetyczny może zostać wykorzystany do znalezienia rozwiązania problemów w czasie szybszym niż metody analityczne. Również w takich w których zastosowanie metod analitycznych może być utrudnione lub niemożliwe. Algorytmy genetyczne mają również wady. Nie zawsze znajdują one najbardziej optymalne rozwiązanie, ale zazwyczaj wynik ten będzie satysfakcjonujący.

Nawet dla tak prostego problemu jak ten będący przedmiotem tego ćwiczenia nie udało się uzyskać konsekwencji w odnajdywaniu najlepszego możliwego rozwiązania mimo zwiększania liczby generacji do 10000 (dalsze zwiększanie uznane zostało nieuzasadnione czasowo). Jednak zawsze było ono dość dobre (wartość najgorszego elementu populacji nie mniejsza niż 120 już przy 100 generacjach).