Zadanie projektowe nr 1 – Nonogram

Autor: Marcin Belicki

1. Wstęp

   W każdym z algorytmów została wykorzystana biblioteka pyeasyga, odpowiednio zmodyfikowana w ten sposób aby zwracała listę najlepszych osobników oraz ich fitnessów, zgodnie z poniższym kodem.

```
def run(self):
    """Run (solve) the Genetic Algorithm."""
    self.create_first_generation()
    najlepsi=[]
    fitnesy=[]
    #while self.current_generation[0].fitness != 10:
    for _ in range(self.generations):
        najlepsi.append(self.best_individual()[1]) # zapisywanie
najlepszego osobnika
        fitnesy.append(self.best_individual()[0]) # zapisywanie fitnessu
najlepszego osobnika
        self.create_next_generation()
        print(_)
    najlepsi.append(self.best_individual()[1])
    fitnesy.append(self.best_individual()[0])
    return [fitnesy,najlepsi]
```

2. Przetestowane algorytmy

   2.1. Algorytm sprawdzający zgodność kolumn i wierszy

   Jest to najprostsza forma tego algorytmu – zamienia on chromosom (wektor o wartościach „0” i „1”) na macierz o odpowiednich wymiarach za pomocą funkcji reshape. Następnie wartości w kolumnach i wierszach są odpowiednio zliczane, tak aby zostały one zakodowane w taki sposób jak rozwiązywany nonogram. Następnie każda kolumna i każdy wiersz są porównywane z danymi wejściowymi. Jeśli dana kolumna/wiersz jest równa danym wejściowym to wartość fitness zwiększa się o 1. W przeciwnym wypadku wartość nie zwiększa się. Maksymalna liczba punktów możliwych do zdobycia jest równa długości chromosomu który jest równy iloczynowi liczby wierszy i liczby kolumn danych wejściowych.

   Poniżej zostały przedstawione poszczególne fragmenty kodu (szczegóły zostały ujęte w formie komentarzy).

```
def tonono(wej): # funkcja konwertująca macierz bitów na format nonogramu
    wyj=[]
    for i in range(len(wej)):
        rzad = []
        licz = 0
        for j in range(len(wej[0])):
            if wej[i][j] == 1:
                licz = licz + 1
            else:
                if licz != 0:
                    rzad.append(licz)
                licz = 0
        if licz != 0:
            rzad.append(licz)
        wyj.append(rzad)
    return wyj # wynikiem jest tablica zawierająca dwie podtablice, gdzie w
każdej zawarty jest wektor ciągów wartości "1" - w tablicy o indeksie 0 dla
wierszy, w tablicy o indeksie 1 dla kolumn
```

```
def roznica(x,y): # funkcja obliczająca różnicę dwóch wektorów
(niekoniecznie równych długości)
    wynik=[]
    if len(x)>len(y): # jeśli któryś z wektorów jest krótszy to dopisywana
jest do niego odpowiednia liczba zer
       y=np.append(y,np.zeros(len(x)-len(y)))
    if len(x)<len(y):
        x=np.append(x, np.zeros(len(y) - len(x)))
    for i in range(len(x)):
        wynik.append(abs(x[i]-y[i]))
    return sum(wynik)
```

```
def fitness1(chromosom, data):  # funkcjka fitness1 - zliczająca poprawne
kolumny i wiersze
    mac_chr = np.array(chromosom).reshape(len(data[0]), len(data[1]))
#konwersja chromosomu na macierz
    pion = tonono(mac_chr) # wyznaczanie odpowiednich wektorów
(analogicznych do tych w nonogramie) dla wszystkich wierszy
    poziom = tonono(np.transpose(mac_chr)) # wyznaczanie odpowiednich
wektorów (analogicznych do tych w nonogramie) dla wszystkich kolumn
    wynik = []
    for i in range(len(pion)):
        wynik.append(roznica(pion[i], data[0][i])==0)
    for i in range(len(poziom)):
        wynik.append(roznica(poziom[i], data[1][i])==0)
    return sum(wynik)
```

```
ga = pyeasyga.GeneticAlgorithm(data, # ustalanie parametrów algorytmu
genetrycznego ga
                               population_size=200,
                               generations=500,
                               crossover_probability=0.8,
                               mutation_probability=1,
                               elitism=False,
                               maximise_fitness=True)
```

```
def create_individual(data): #funkcja generująca pierwsze pokolenie
    chromosom = []
    for i in range(0, len(data[0]) * len(data[1])):
        chromosom.append(np.random.choice([0, 1]))
    return chromosom
```

```
print(create_individual(data))
ga.create_individual = create_individual
ga.fitness_function = fitness1
[fitnesy,best] = ga.run() #inicjacja algorytmu genetycznego
print(best)
plt.pcolor(
np.flip(np.array(ga.best_individual()[1]).reshape(len(data[0]),len(data[1])
),0) , cmap = 'Greys' ) #rysowanie najleszego rozwiązania
plt.axis('equal')
plt.show()
fig = plt.figure()
```

2.2. Algorytm obliczający różnicę kolumn i wierszy

Algorytm jest w swoich założeniach podobny do algorytmu z punktu 2.1. Chromosom ma również postać wektora o wartościach „0” i „1”, oraz wartości te są odpowiednio przez funkcję fitness zliczane analogicznie do tworzenia nonogramu.

Jednakże wartości nie są bezpośrednio porównywane z wartościami z danych wejściowych, zamiast tego dla wektorów każdej kolumny oraz każdego wiersza obliczany jest moduł z różnicy między poszczególnymi wyrazami danych wejściowych oraz danych pochodzących z chromosomu. Jeśli wektory te są różnej długości to algorytm dopisuje odpowiednią liczbę zer do krótszego z nich. Następnie wszystkie tak powstałe moduły różnic są sumowane. Im większa suma tym dalej od prawidłowego rozwiązania, dlatego algorytm dąży do osiągniecia wartości fitness równej 0.

Poniżej zostały przedstawione poszczególne fragmenty kodu – różni się on od poprzedniego jedynie zastosowaną funkcją fitness.

```
def fitness2 (chromosom,data): # funkcja fitness2 - sumującego różnice
miedzy wektorami chromosomu a danymi wejściowymi
    mac_chr = np.array(chromosom).reshape(len(data[0]),len(data[1]))
    pion = tonono(mac_chr)
    poziom = tonono(np.transpose(mac_chr))
    wynik=[]
    for i in range(len(pion)):
        wynik.append(roznica(pion[i],data[0][i]))
    for i in range(len(poziom)):
        wynik.append(roznica(poziom[i], data[1][i]))
    return -dlugosc_euklidesowa(wynik)
```

2.3. Algorytm sprawdzający kombinację odstępów

Algorytm koduje rozwiązanie za pomocą odstępów pomiędzy ciągami wartości „1” w wierszach, w taki sposób, że każdej wartości danych wejściowych w wierszach przypisywana jest odpowiednia liczba. Jeśli wartość odstępu jest równa 0 to w takim wypadku, odstęp między najbliższym ciągiem „1” po lewej stronie wynosi 1 kratkę, natomiast jeśli 0 wystąpi w przypadku pierwszego ciągu, to znajduje się on bezpośrednio przy krawędzi macierzy.

Algorytm jest o tyle skomplikowanym, że nie działa on bezpośrednio. Chromosomy mają postać ciągu odpowiednich odstępów, jednakże ich fitness jest obliczany za pomocą innego algorytmu genetycznego, którego rozwiązaniem jest ciąg wartości „-1”, „0”, i „1”, którego długość odpowiada długości pierwotnego chromosomu. Drugi algorytm genetyczny działa w ten sposób, że dodaje do pierwotnego chromosomu odpowiednie wartości „-1”, „0” lub „1” i następnie rozkodowuje rozwiązanie za pomocą odpowiedniej funkcji na macierz „0” i „1”, a tak otrzymana macierz jest porównywana z danymi wejściowymi analogicznie jak zostało to przedstawione w punkcie 2.2. Po uzyskaniu najlepszego osobnika z drugiego algorytmu, jego wartości są odpowiednio dodawane do pierwotnego chromosomu i w tej postaci przechodzi on do nowego pokolenia. Algorytm analogicznie jak w poprzednim przypadku dąży do osiągnięcia wartości fitness równej 0.

Poniżej zostały przedstawione poszczególne fragmenty kodu. Jego największa różnica zawarta jest w funkcjach fitness oraz wykorzystaniu funkcji zamieniających chromosom na macierz rozwiązania.

```
def makefrom(k,l,dl): # funkcja tworząc wektor o długości dl składający się
z zer i jedynek z podanymi ciągami k oraz odstępami l
    wyn=np.zeros(dl)
    m=0
    for i in range(len(k)):
        for j in range(l[i]):
            m=m+1
        for j in range(k[i]):
            if m < dl:
                wyn[m]=1
```

```
            m= m + 1
        m=m+1
    return wyn
```

```
def zamien(chromosom,data): # funkcja konwertująca wektor odstępów
(chromosom na macierz zer i jedynek)
    dlugosc = len(data[1])
    mac_chr=[]
    od = 0
    do = len(data[0][0])
    for i in range(len(data[0])):
        mac_chr.append(makefrom(data[0][i], chromosom[od:do], dlugosc))
        od = od + len(data[0][i])
        if i < len(data[0]) - 1:
            do = do + len(data[0][i + 1])
    return mac_chr
```

```
def fitness1 (chromosom,data): # funkja fitness wykorzystana w algorytmie
ga - jej chromosomami są wektory zawierające wartości -1, 0 i 1
    global test
    wyn=[]
    for i in range(len(test)):
        wyn.append(test[i]+ga.best_individual()[1][i])
    # print(wyn)
    for i in range(len(test)):
        wyn[i] =int( test[i] + chromosom[i])
        if wyn[i] < 0:
                wyn[i]=0
    mac_chr=zamien(wyn,data)
    poziom = tonono(np.transpose(mac_chr))
    dodatnie = []
    pion = tonono(mac_chr)
    for i in range(len(pion)):
        dodatnie.append(sum(roznica(pion[i], data[0][i])))
    for i in range(len(poziom)):
        dodatnie.append(sum(roznica(poziom[i], data[1][i])))
    return -dlugosc_euklidesowa(dodatnie)
```

```
def fitness2 (chromosom,data): # funkcja fitness głównego algorytmu ga2,
której chromosomami są wartości odstępów pomiędzy ciągami "1" w wierszach
    print(chromosom)
    global test
    test = chromosom
    najlepszy=ga.run() # inicjacja algorytmu genetycznego ga
    for i in range(len(test)):
        test[i] = (test[i] + ga.best_individual()[1][i])
        if test[i] < 0:
            test[i] = 0
    return [ga.best_individual()[0],test] # funkcja zwraca również wartość
chromosomu zsumowanego z najlepszym osobnikiem algorytmu genetycznego ga,
wynik tego działania jest przypisywany temu chromosomowi

ga = pyeasyga.GeneticAlgorithm(data, #parametry algorytmu ga
                               population_size=50,
                               generations=50,
                               crossover_probability=0.8,
                               mutation_probability=1,
                               elitism=False,
                               maximise_fitness=True)
```

```
def create_individual(data): # funkcja generująca pierwsze pokolenie
algorymu ga
    chromosom = []
    liczba=0
    for i in range(len(data[0])):
        liczba=liczba+len(data[0][i])
    for i in range(liczba):
        chromosom.append(np.random.choice([-1,0,1]))
    return chromosom
```

```
ga2 = pyeasyga2.GeneticAlgorithm(data, #parametry algorytmu ga2
                                 population_size=50,
                                 generations=20,
                                 crossover_probability=0.8,
                                 mutation_probability=0.05,
                                 elitism=True,
                                 maximise_fitness=True)
```

```
def create_individual2(data): # funkcja generująca pierwsze pokolenie
algorymu ga2
    chromosom = []
    liczba=0
    for i in range(len(data[0])):
        liczba=liczba+len(data[0][i])
    for i in range(liczba):
        chromosom.append(np.random.choice([1]))
    return chromosom
ga2.create_individual = create_individual2
ga2.fitness_function = fitness2
ga2.run()  #inicjacja algorytmu ga2
```

3. Generowanie inputów

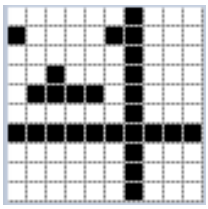
   W celu łatwiejszej możliwości zmieniania inputów (jak choćby w programie Paint) algorytm odczytuje input z pliku bmp za pomocą kodu.

```
im = Image.open('orzel.bmp') # import obrazu za pomocą biblioteki PIL
p = np.array(im) # konwertowanie obrazu na macierz z wartościami RGB
def bitconvertion(wej): # funkcja konwertowująca macierz z danymi RGB na
macierz bitową
    wyj = np.zeros((len(wej), len(wej[0])))
    for i in range(len(wej[0])):
        for j in range(len(wej)):
            if np.mean(wej[j][i]) < 50: # jeśli średnia wartość piksela
(j,i) edzie mniejsza niż 50 to wartość jest równa 1
                wyj[j][i] = 1
    return wyj
```

```
# tworzenie danych wejściowych
obraz = bitconvertion(p)
data=[tonono(obraz),tonono(np.transpose(obraz))]
print(data)
```

4. Optymalizacja algorytmów genetycznych

Analiza została przeprowadzona dla danych wejściowych 10x10, których rozwiązaniem jest:



Dane wejściowe:

[[[1], [1, 2], [1], [1, 1], [4, 1], [1], [10], [1], [1], [1]], [[1, 1], [1, 1], [2, 1], [1, 1], [1, 1], [1, 1], [10], [1], [1], [1]]]

4.1. Algorytm sprawdzający zgodność kolumn i wierszy

4.1.1.Próba porównawcza

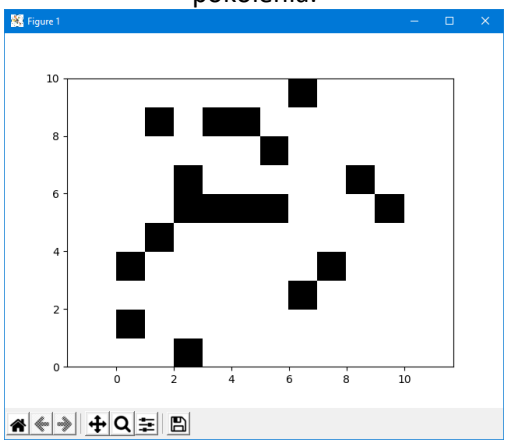
Populacja: 200

Generacje: 500

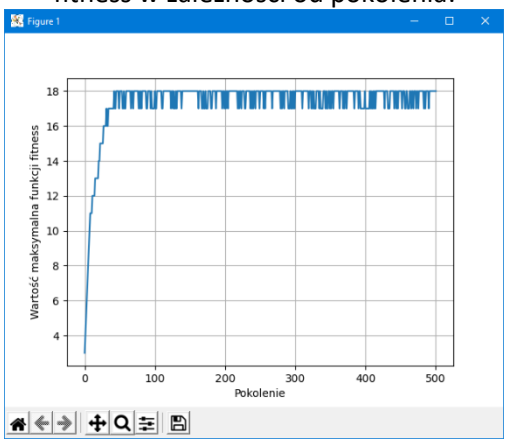
Prawdopodobieństwo mutacji: 1

Czas wykonywania: 27,415 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

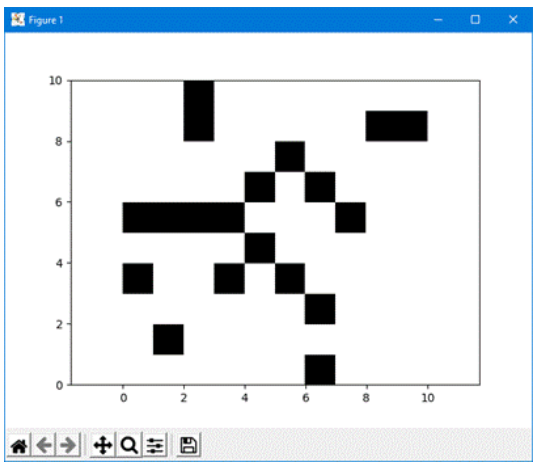


4.1.2.Zmiana liczby populacji

Populacja: 100 (pozostałe parametry bez zmian względem próby porównawczej)ś

Czas wykonywania: 13,879 s

Najlepsze rozwiązanie ostatniego pokolenia:



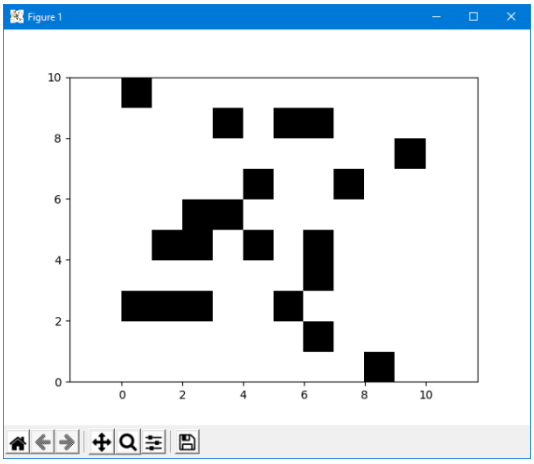
Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

4.1.3.Zmiana prawdopodobieństwa mutacji

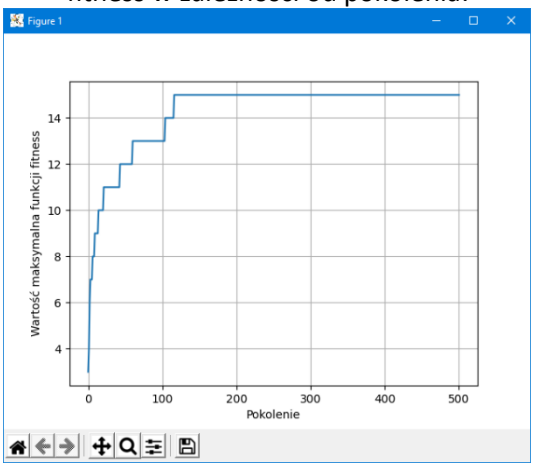
Prawdopodobieństwo mutacji: 0,5 (pozostałe parametry bez zmian względem próby porównawczej)

Czas wykonywania: 29,319 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:



4.2. Algorytm obliczający różnicę kolumn i wierszy

4.2.1.Próba porównawcza

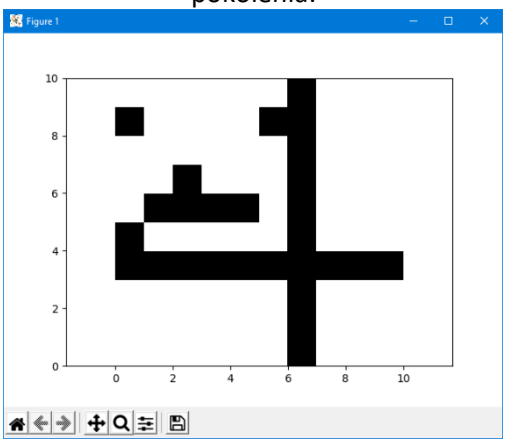
Populacja: 200

Generacje: 500

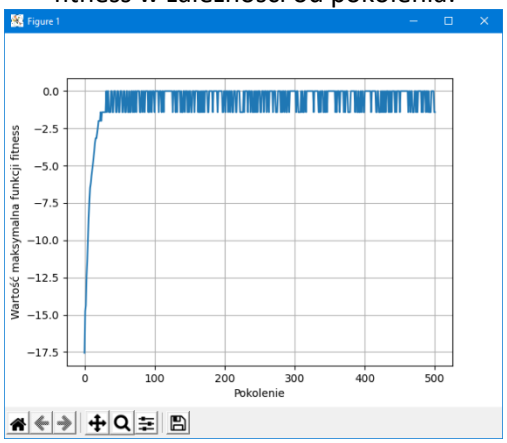
Prawdopodobieństwo mutacji: 1

Czas wykonywania: 23,984 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

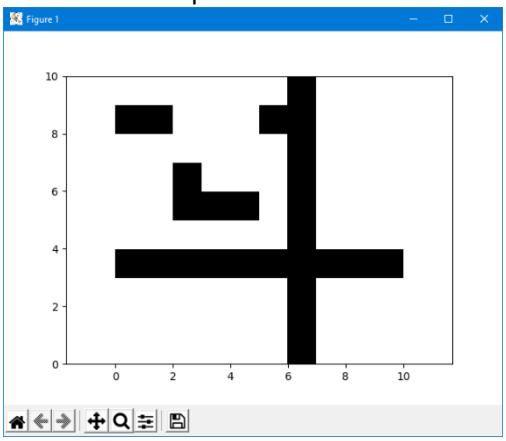


4.2.2.Zmiana liczby populacji

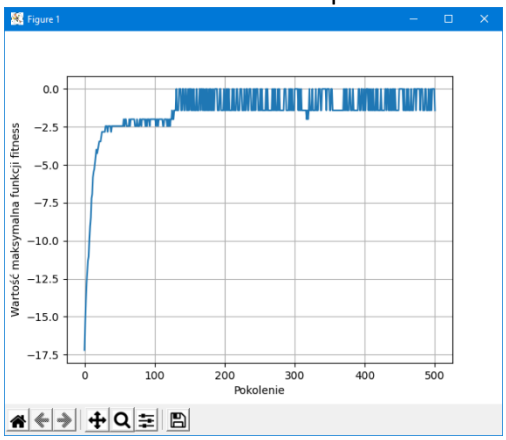
Populacja: 100 (pozostałe parametry bez zmian względem próby porównawczej)

Czas wykonywania: 12,177 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

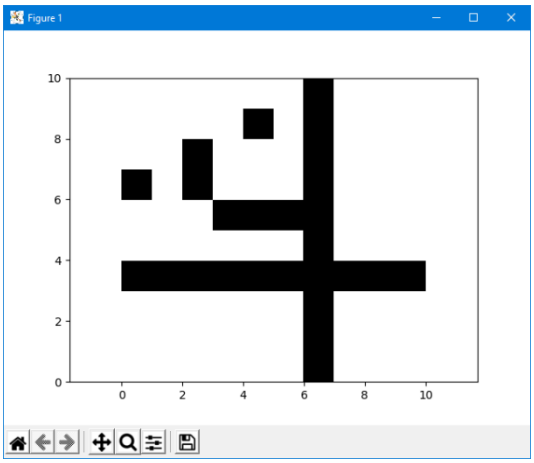


4.2.3.Zmiana prawdopodobieństwa mutacji

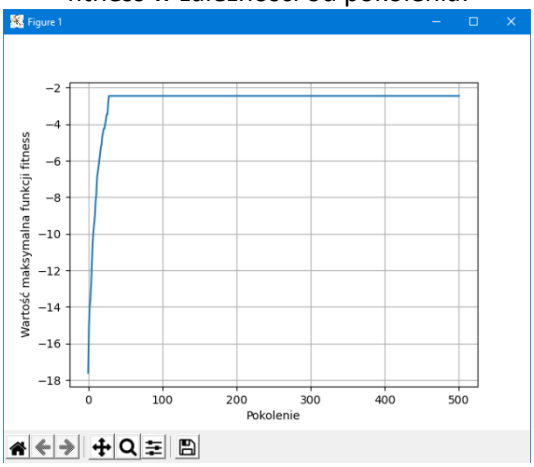
Prawdopodobieństwo mutacji: 0,5 (pozostałe parametry bez zmian względem próby porównawczej)

Czas wykonywania: 26,806 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:



4.3. Algorytm sprawdzający kombinację odstępów

4.3.1. Próba porównawcza

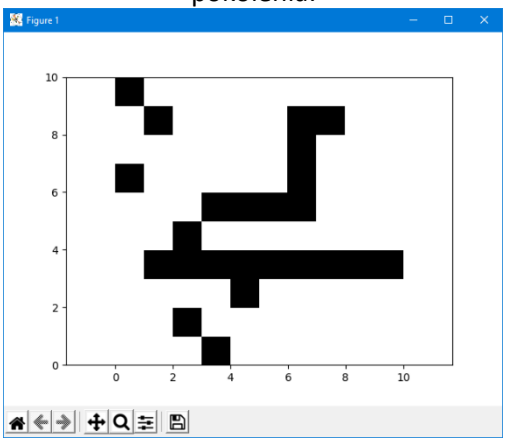
Populacja: 10

Generacje: 10

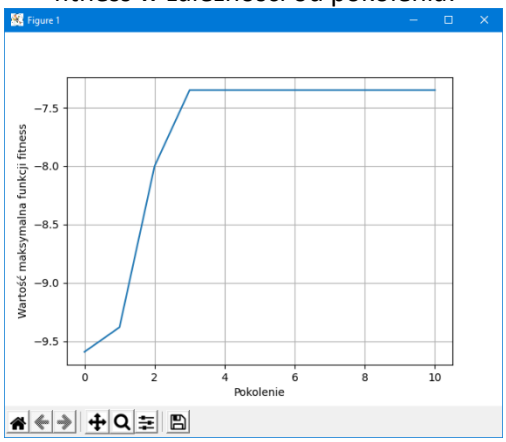
Prawdopodobieństwo mutacji: 0,05

Czas wykonywania: 99,265 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

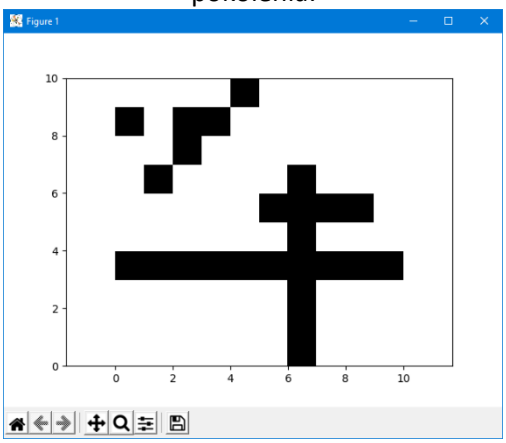


4.3.2. Zmiana liczby populacji

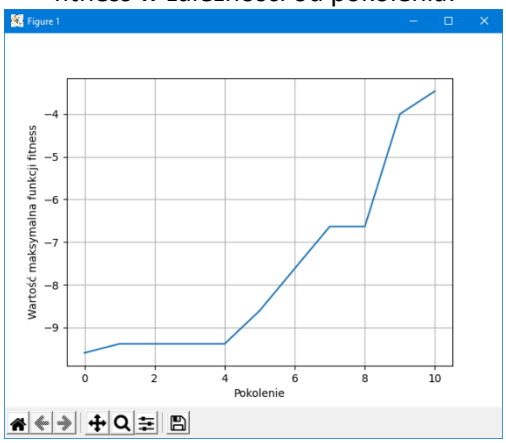
Populacja: 5 (pozostałe parametry bez zmian względem próby porównawczej)

Czas wykonywania: 49,522 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:

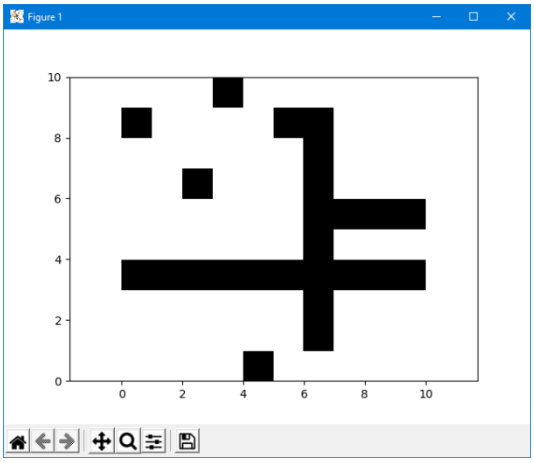


4.3.3.Zmiana prawdopodobieństwa mutacji

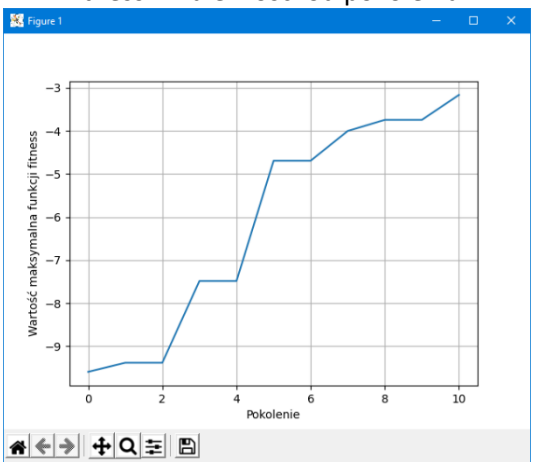
Prawdopodobieństwo mutacji: 0,1 (pozostałe parametry bez zmian względem próby porównawczej)

Czas wykonywania: 99,384 s

Najlepsze rozwiązanie ostatniego pokolenia:



Wykres maksymalnej wartości funkcji fitness w zależności od pokolenia:



5. Wnioski

Najbardziej efektywnym okazał się algorytm opisany w punkcie 2.2, dla populacji 200 daje on poprawne rozwiązanie (fitnessmax=0) już w około 50 pokoleniu. Jego czas jest również najkrótszy, jednakże jest tak ze względu na fakt iż w algorytmie opisanym w punkcie 2.1 zastosowano funkcję różnica, w celu porównania wektorów, gdyby zmodyfikować algorytm tak, że wektory porównywane byłyby bezpośrednio skróciłoby to czas wykonywania.

Z pozytywnym rezultatem spotkała się duża wartość prawdopodobieństwa mutacji (aż 100%) w dwóch pierwszych algorytmach. Może tak być, ze względu na fakt iż mutacja może pozytywnie wpłynąć na poprawną zmianę pojedynczego bitu, co skutkuje czasami zwiększeniem funkcji fitness.

Z przeprowadzonych prób można wysnuć wniosek iż do rozwiązania nonogramu 10x10 wystarczy zastosować algorytm opisany w punkcie 2.2 z populacją 100.

Najmniej efektywnym okazał się algorytm nr 3, który pomimo najdłuższego wykonywania nie otrzymywał poprawnego rozwiązania.