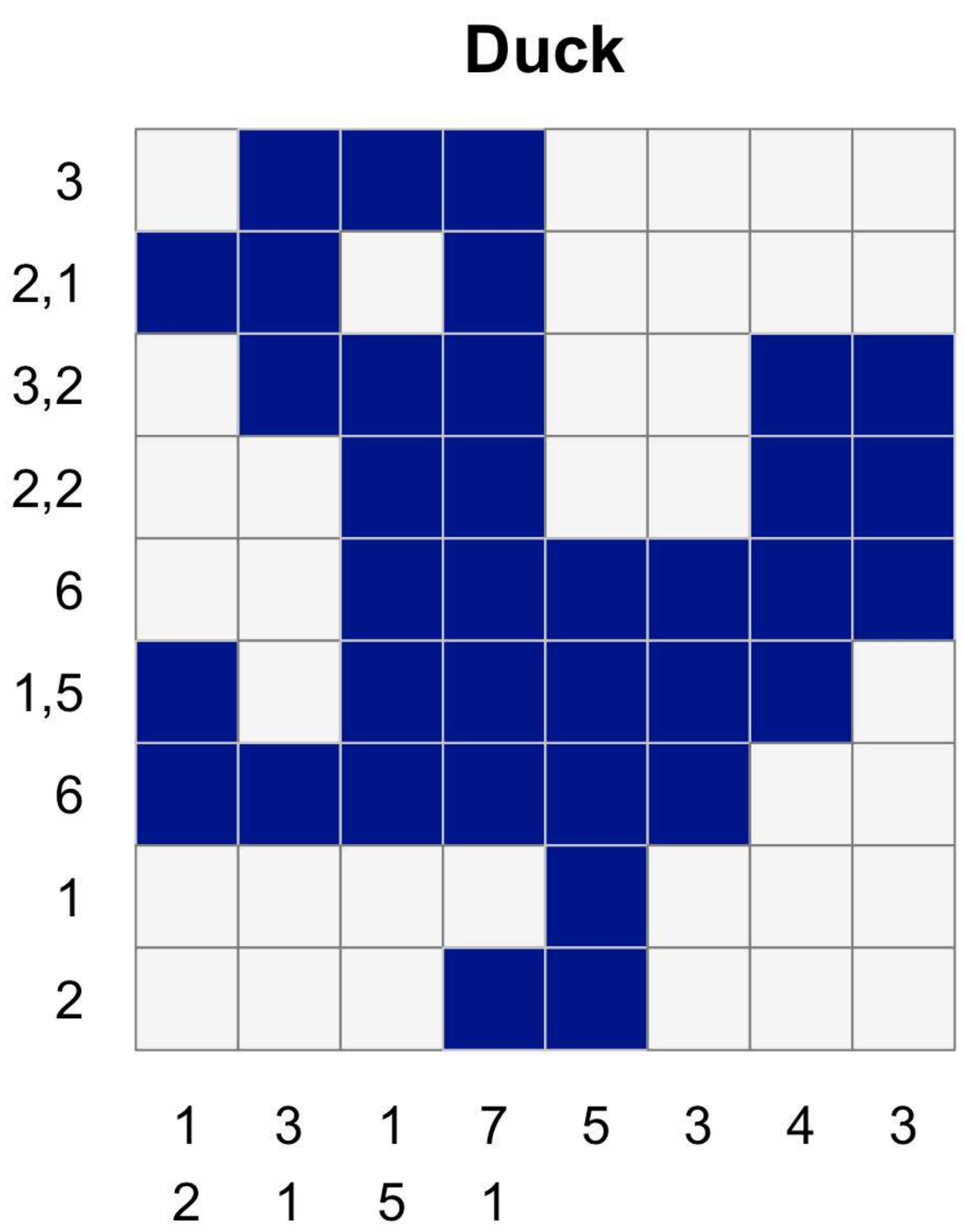
Rozwiazywanie nonogramow

Algorytym genetyczny i inteligencja roju

Nonogram to rodzaj zagadki, w której należy zamalować niektóre kratki na czarno tak, by powstał z nich obrazek. By odgadnąć, które kratki należy zamalować, należy odszyfrować informacje liczbowe przy każdym wierszu i kolumnie kratek obrazka.



Implementacja

Do rozwiązania problemu utworzyłem klasę Nonogram. Ważnym wymaganiem jest to, aby utworzone Nonogramy były kwadratowe (równa liczba kolumn i rzędów). Jako wartości kolumn i rzędów wpisujemy liczbę zamalowanych kratek. Przykład utworzenia nonogramu:

```
non_duck = Nonogram(
    [[1, 2], [3, 1], [1, 5], [7, 1], [5], [3], [4], [3], [0]],
    [[3], [2, 1], [3, 2], [2, 2], [6], [1, 5], [6], [1], [2]],
)
```

```
class Nonogram:
    def __init__(self, columns, rows, sol=None):
        self.columns = columns
        self.rows = rows
        self.sol = sol if sol is not None else self.generate_empty_solution()

    def __str__(self):
        return f"Columns: {self.columns}\nRows: {self.rows}\nSolution: {self.sol}"
```

funkcje Fitness

szybki opis działania funkcji

“fitness”

Pierwsza funkcja fitness. Funkcja “fitness” porównuje rozwiązanie wygenerowane przez algorytm genetyczny z oczekiwanym rozwiązaniem nonogramu. Dla każdego rzędu i kolumny sprawdza, czy ciągi jedynek odpowiadają wymaganym blokom (sekwencjom jedynek). Przykładowo dla rzędu [0,1,1,0,1] zostanie utworzona lista [2,1]. Naliczane są kary za każde odstępstwo od oczekiwanego rozwiązania.

```
def fitness(ga_instance, solution, solution_idx):
    resultRow = []
    resultCol = []
    solution = [
        solution[i : i + len(self.rows)]
        for i in range(0, len(solution), len(self.rows))
    ]
    for i in solution:
        count = 0
        tmp = []
        for j in i:
            if j == 1:
                count += 1
            elif count != 0:
                tmp.append(count)
                count = 0
        if count != 0:
            tmp.append(count)
        resultRow.append(tmp)
    for i in range(len(solution[0])):
        count = 0
        tmp = []
        for j in range(len(solution)):
            if solution[j][i] == 1:
                count += 1
            elif count != 0:
                tmp.append(count)
                count = 0
        if count != 0:
            tmp.append(count)
        resultCol.append(tmp)
    wrong = 0
    for i in range(len(self.rows)):
        if not arrayEquals(self.rows[i], resultRow[i]):
            wrong -= 1
        if not arrayEquals(self.columns[i], resultCol[i]):
            wrong -= 1
    return wrong
```

“fitness advanced”

Funkcja fitness_advanced działa podobnie do fitness, ale dodatkowo nalicza większe kary za różnice w liczbie bloków jedynek oraz za błędy w ich pozycjach. W przypadku brakujących wierszy lub kolumn, naliczane są dodatkowe kary.

```
def fitness_advanced(ga_instance, solution, solution_idx):
    resultRow = []
    resultCol = []
    solution = [
        solution[i : i + len(self.rows)]
        for i in range(0, len(solution), len(self.rows))
    ]

    # Calculate row blocks
    for i in solution:
        count = 0
        tmp = []
        for j in i:
            if j == 1:
                count += 1
            else:
                if count != 0:
                    tmp.append(count)
                    count = 0
        if count != 0:
            tmp.append(count)
        resultRow.append(tmp)

    # Calculate column blocks
    for i in range(len(solution[0])):
        count = 0
        tmp = []
        for j in range(len(solution)):
            if solution[j][i] == 1:
                count += 1
            else:
                if count != 0:
                    tmp.append(count)
                    count = 0
        if count != 0:
            tmp.append(count)
        resultCol.append(tmp)

    wrong = 0

    # Evaluate rows
    for i in range(len(self.rows)):
        if i >= len(resultRow):
            wrong -= 10 * abs(len(self.rows[i]))  # Penalty for missing rows
            continue
        if len(self.rows[i]) != len(resultRow[i]):
            wrong -= abs(len(self.rows[i]) - len(resultRow[i])) * 10  # Larger penalty for block
count difference
        min_len = min(len(self.rows[i]), len(resultRow[i]))
        for j in range(min_len):
            wrong -= abs(self.rows[i][j] - resultRow[i][j])  # Penalty for length mismatch
        # Extra penalty for position mismatch
        for j in range(min_len):
            if self.rows[i][j] != resultRow[i][j]:
                wrong -= 1

    # Evaluate columns
    for i in range(len(self.columns)):
        if i >= len(resultCol):
            wrong -= 10 * abs(len(self.columns[i]))  # Penalty for missing columns
```

“fitness_another”

Funkcja fitness_another buduje tablicę binarną na podstawie rozwiązań genetycznych. Następnie porównuje liczbę bloków jedynek w każdym wierszu i kolumnie z oczekiwanym wzorem. Naliczane są kary za każde odstępstwo od wymagań nonogramu, zarówno w liczbie, jak i długości bloków jedynek.

```
def fitness_another(ga_instance, solution, solution_idx):
    tab = [[0 for _ in range(len(self.columns))] for _ in range(len(self.rows))]
    iterator = 0
    wrong = 0
    solution = solution.astype(int)

    # Process rows
    for i in range(len(self.rows)):
        for j in range(len(self.rows[i])):
            for k in range(self.rows[i][j]):
                if iterator + k < len(solution):
                    if solution[iterator + k] < len(self.rows):
                        tab[i][solution[iterator + k]] = 1
                    else:
                        wrong -= 1
                else:
                    wrong -= 1
            iterator += 1

    # Process columns
    for i in range(len(self.columns)):
        for j in range(len(self.columns[i])):
            for k in range(self.columns[i][j]):
                if iterator + k < len(solution):
                    if solution[iterator + k] < len(self.columns):
                        tab[solution[iterator + k]][i] = 1
                    else:
                        wrong -= 1
                else:
                    wrong -= 1
            iterator += 1

    resultRow = []
    resultCol = []

    # Process result rows
    for i in tab:
        counter = 0
        temp = []
        for j in i:
            if j == 1:
                counter += 1
            elif counter != 0:
                temp.append(counter)
                counter = 0
        if counter != 0:
            temp.append(counter)
        resultRow.append(temp)

    # Process result columns
    for i in range(len(tab[0])):
        counter = 0
        temp = []
        for j in range(len(tab)):
            if tab[j][i] == 1:
                counter += 1
            elif counter != 0:
                temp.append(counter)
                counter = 0
        if counter != 0:
```

“best_fit”

Funkcja best_fit tworzy tablicę binarną na podstawie wektora rozwiązań. Następnie porównuje liczby bloków jedynek w kolumnach z wymaganiami nonogramu. Naliczane są kary za każde odstępstwo od oczekiwanego wzoru bloków jedynek, koncentrując się głównie na kolumnach.

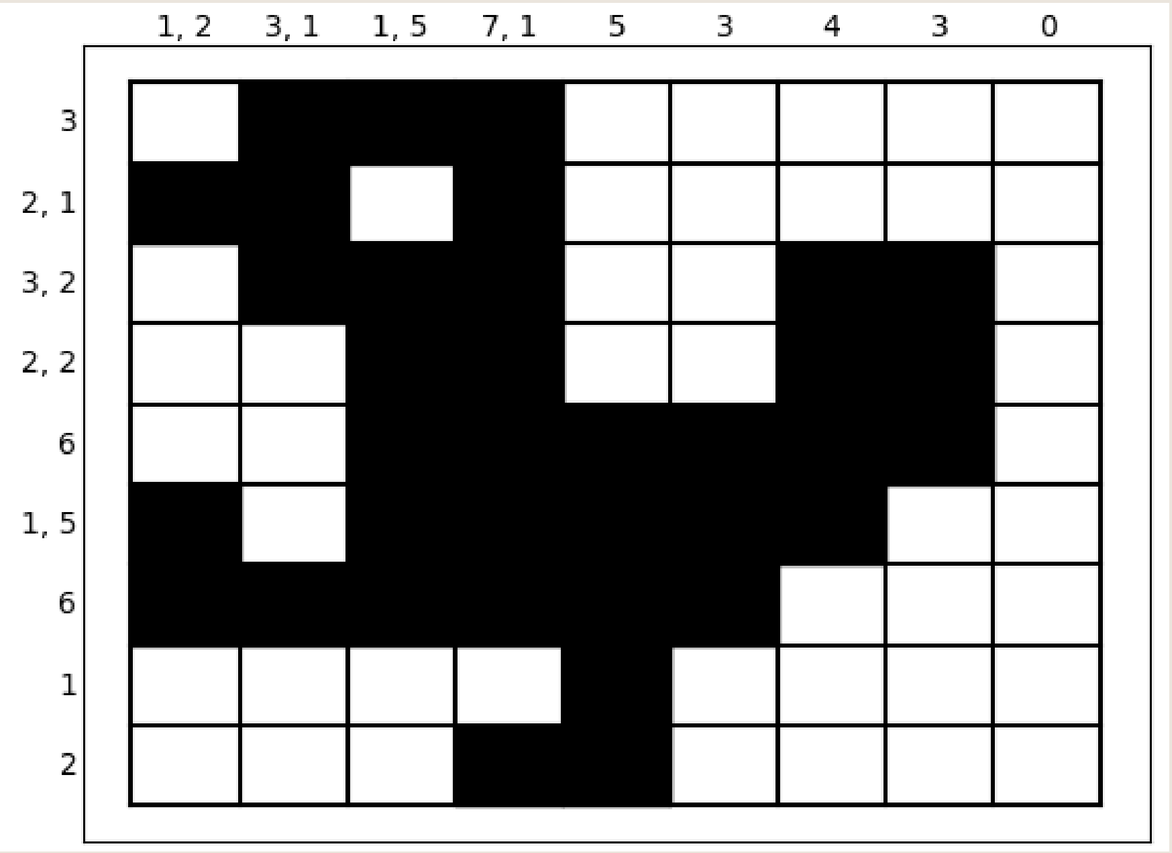
```
def best_fit(ga_instance, solution, solution_idx):
            tab = [[0] * len(self.columns) for _ in range(len(self.rows))]
            iterator = 0
            wrong = 0
            solution = solution.astype(int)

            for i, row in enumerate(self.rows):
                for j, count in enumerate(row):
                    for k in range(count):
                        pos = solution[iterator] + k
                        if pos < len(self.rows):
                            tab[i][pos] = 1
                        else:
                            wrong -= 1
                    iterator += 1

            resultCol = []
            for col in zip(*tab):
                temp = []
                counter = 0
                for cell in col:
                    if cell == 1:
                        counter += 1
                    elif counter:
                        temp.append(counter)
                        counter = 0
                if counter:
                    temp.append(counter)
                resultCol.append(temp)

            for col, res_col in zip(self.columns, resultCol):
                if len(col) == len(res_col):
                    wrong -= sum(1 for a, b in zip(col, res_col) if a != b)
                else:
                    wrong -= abs(len(col) - len(res_col))
                    if len(col) > len(res_col):
                        wrong -= sum(1 for a, b in zip(col, res_col) if a != b)
                    else:
                        wrong -= sum(1 for a, b in zip(res_col, col) if a != b)

            return wrong
```

Przykładowe rozwiązania

Do uzyskania rozwiązań przykładowych monogramów użyłem funkcji “best_fit” oraz następujące parametry algorytmu genetycznego:

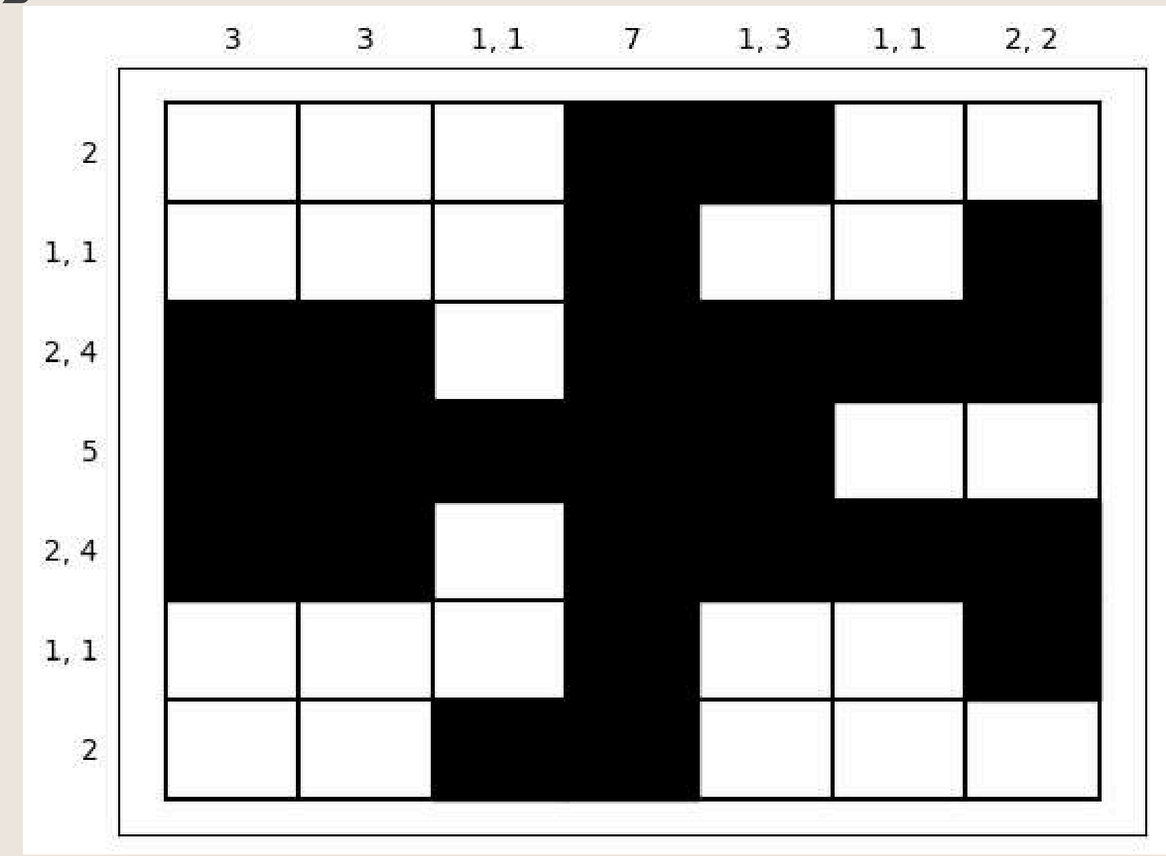
```
fitness_func = "best_fit"
num_generations = 3000
sol_per_pop = 200
mutation_percent_genes = 12
num_parents_mating = 100
keep_parents = 2
parent_type_selection = "sss"
crossover_type = "single_point"
mutation_type = "random"
```

Rezultaty



wyniki

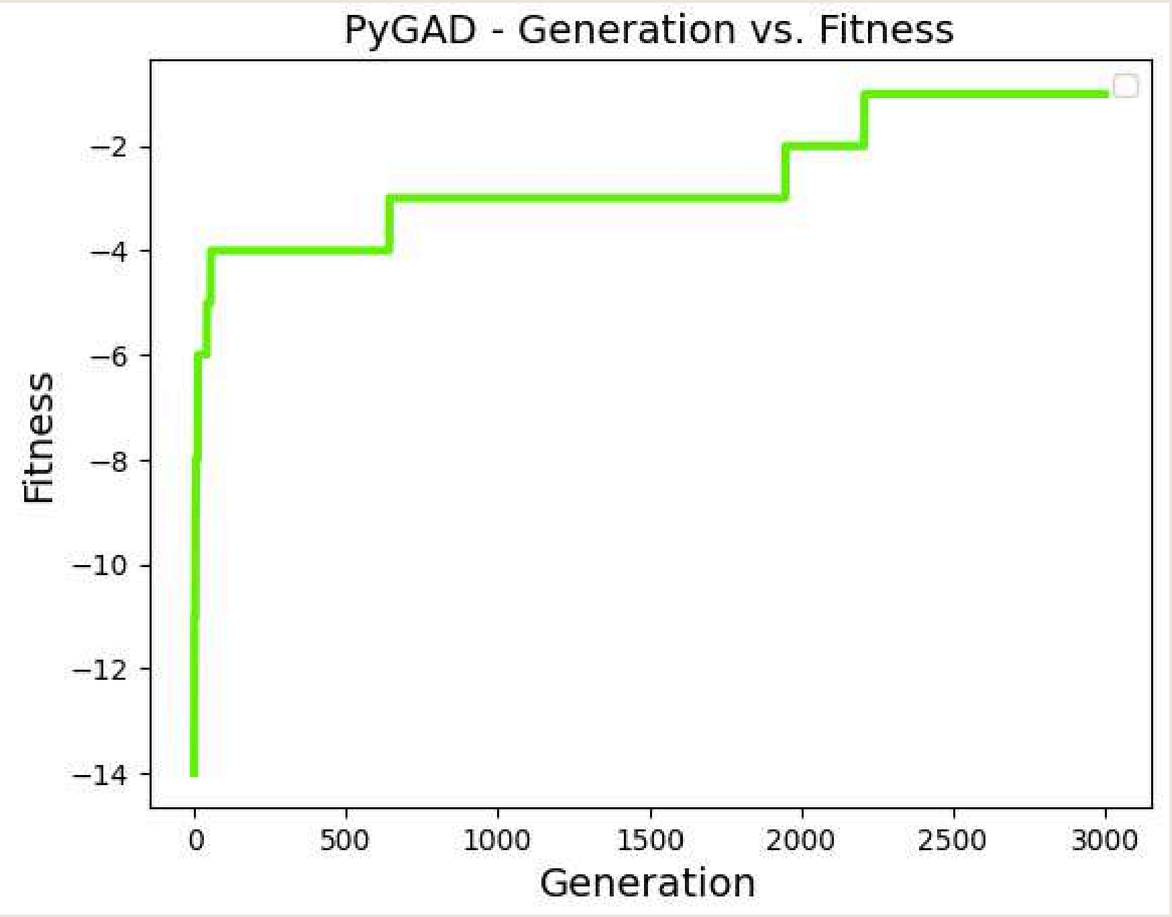
{'**best_solution_generation**': 2210, '**solution_fitness**': -1.0, '**solution**': array([[0, 1, 1, 1, 0, 0, 0, 0, 0], [1, 1, 0, 1, 0, 0, 0, 0, 0], [0, 1, 1, 1, 0, 0, 1, 1, 0], [0, 0, 1, 1, 0, 0, 1, 1, 0], [0, 0, 1, 1, 1, 1, 1, 1, 0], [1, 0, 1, 1, 1, 1, 1, 0, 0], [1, 1, 1, 1, 1, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 1, 0, 0, 0, 0]])}



wyniki

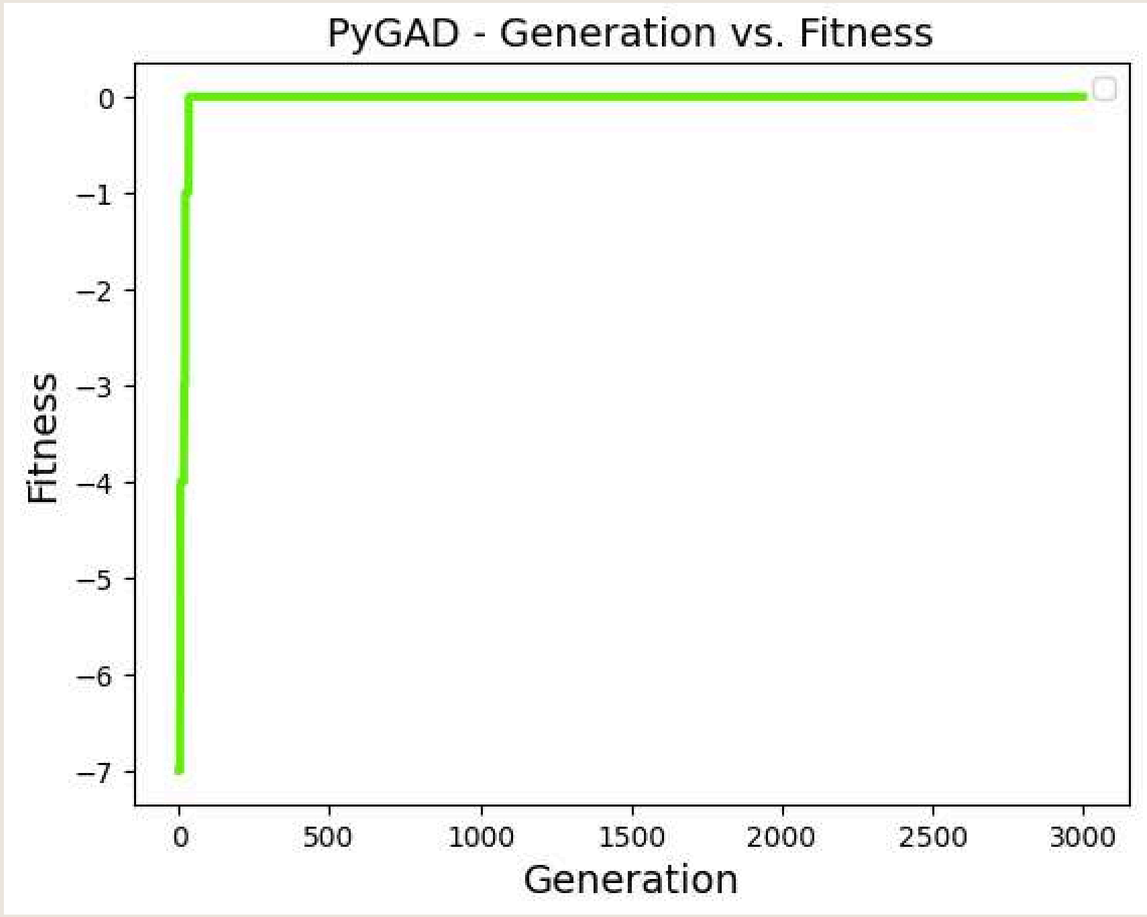
{'**best_solution_generation**': 33, '**solution_fitness**': 0.0, '**solution**': array([[0, 0, 0, 1, 1, 0, 0], [0, 0, 0, 1, 0, 0, 1], [1, 1, 0, 1, 1, 1, 1], [1, 1, 1, 1, 1, 0, 0], [1, 1, 0, 1, 1, 1, 1], [0, 0, 0, 1, 0, 0, 1], [0, 0, 1, 1, 0, 0, 0]])}

Rezultaty - fitness



wyniki - kaczka

{'**best_solution_generation**': 2210, '**solution_fitness**': -1.0, '**solution**': array([1., 0., 3., 1., 6., 6., 2., 2., 0., 2., 0., 4., 3.])}



wyniki - ludzik

{'**best_solution_generation**': 33, '**solution_fitness**': 0.0, '**solution**': array([3., 6., 3., 0., 3., 0., 0., 3., 6., 3., 2.])}

W celu sprawdzenia jak dobrze sobie radzi moja funkcja “**best_fit**” przeprowadziłem test dla 100 rozwiązań dla nonogramu z kaczką, żeby sprawdzić jaka jest potrzebna średnia ilość generacji, oraz ile poprawnych rozwiązań uzyskam. Wyniki prezentują się następująco:

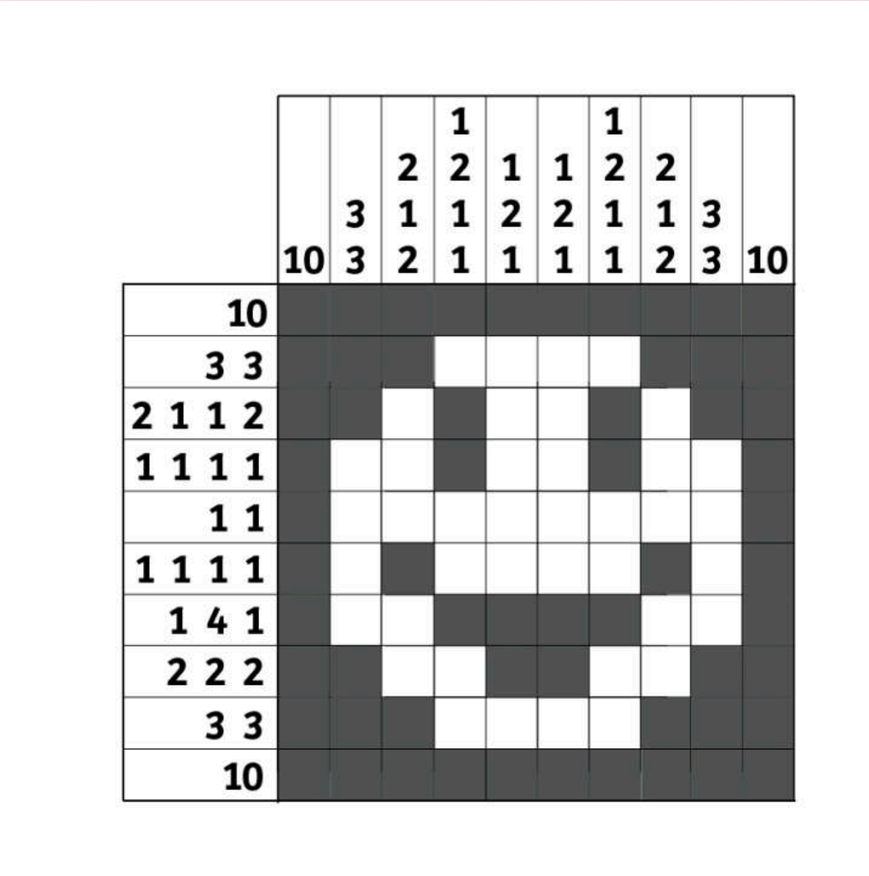
944.32

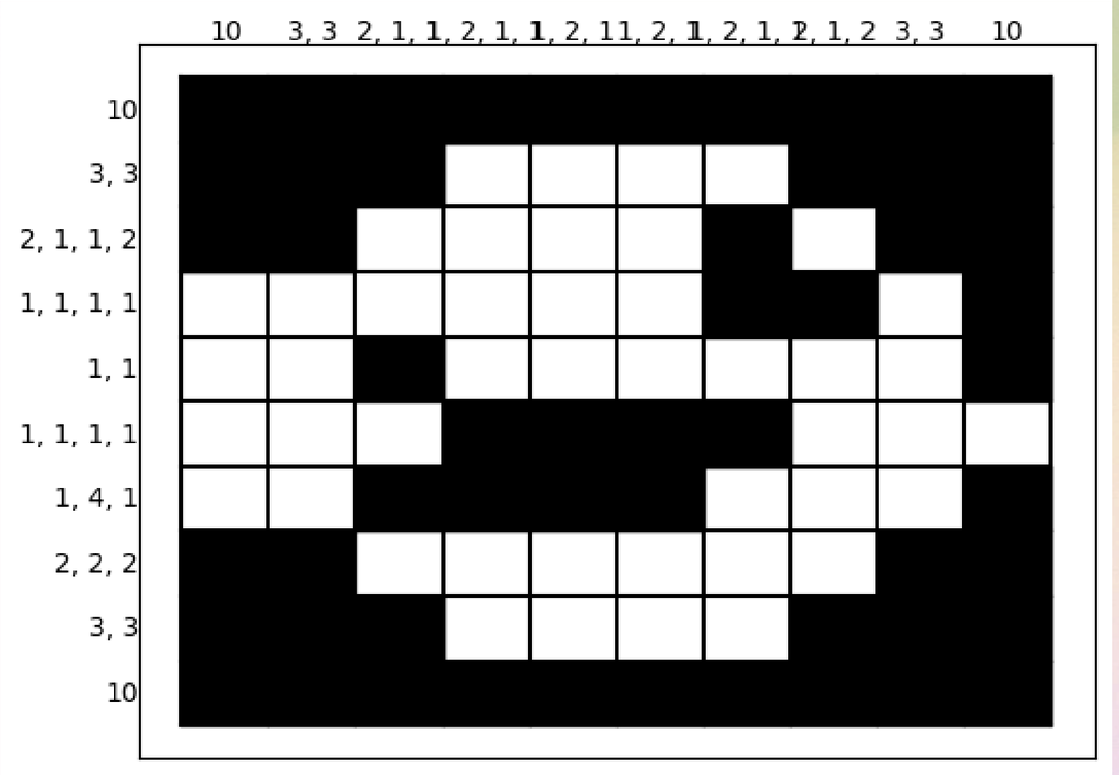
średnia liczba generacji

97%

poprawnych wyników

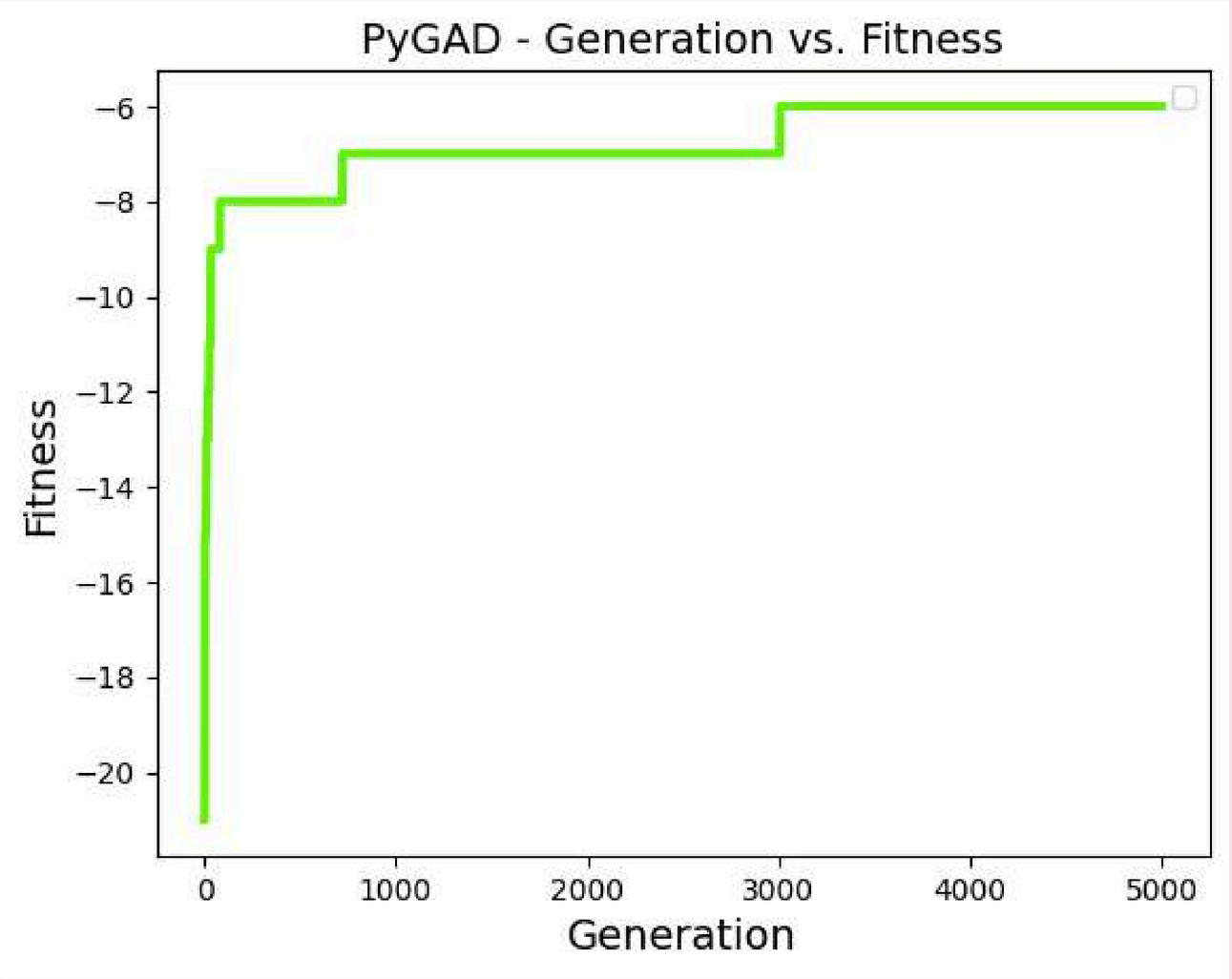
Niestety dla większych nonogramów, algorytm genetyczny nie radzi sobie zbyt dobrze. Przykładowo, po lewej znajduje się docelowy nonogram, który próbowałem rozwiązać, a po prawej wynik mojego algorytmu dla funkcji “best_fit” oraz liczby generacji **5000**.





Wykres fitness

Best solution zostało znalezione dla generacji 3007, później już wynik się nie poprawiał, a czas wykonywania wyniósł 1:07min.

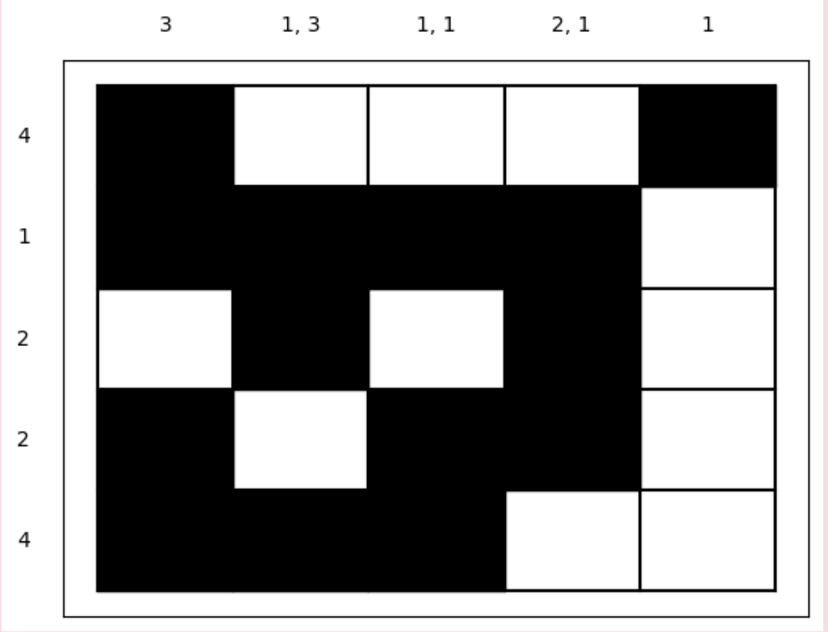


Inteligencja roju

Do testów wykorzystałem takie same funkcje fitness.

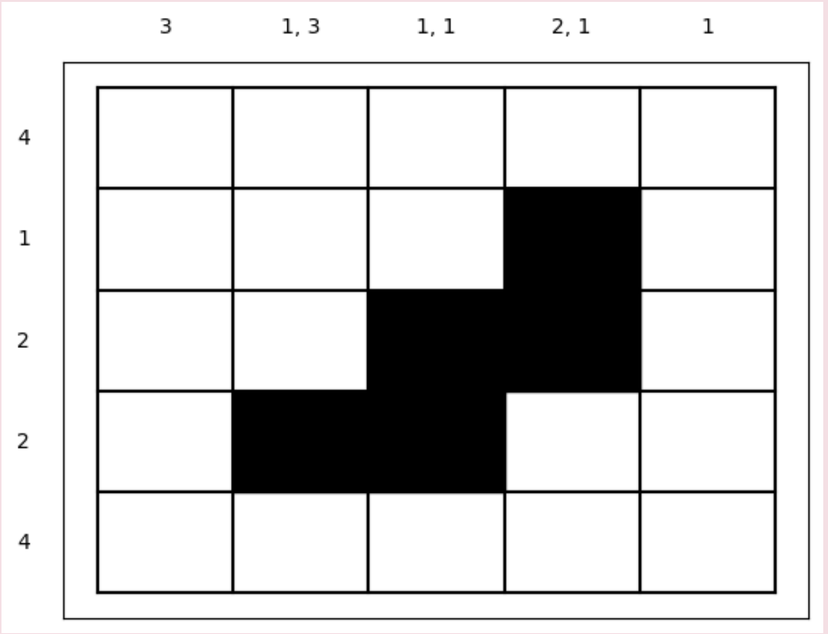
Wyniki

```
dimensions = 1
iters = 1000
options = {'c1': 1.5, 'c2': 0.8, 'w':0.9, 'k': 2, 'p': 2}
```



fitness

```
dimensions = 1
iters = 1000
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9}
```



best fit

Ineligencja roju

Jak widać algorytm genetyczny radził sobie znacznie lepiej niż algorytm PSO. Przeprowadziłem podobny test, w którym uruchomiłem PSO (tym razem dla mniejszego monogramu) 100 razy i poprawny rezultat otrzymałem tylko 3 razy.

Podsumowanie

Dziękuję za uwagę!