Rozwiązywanie nonogramów

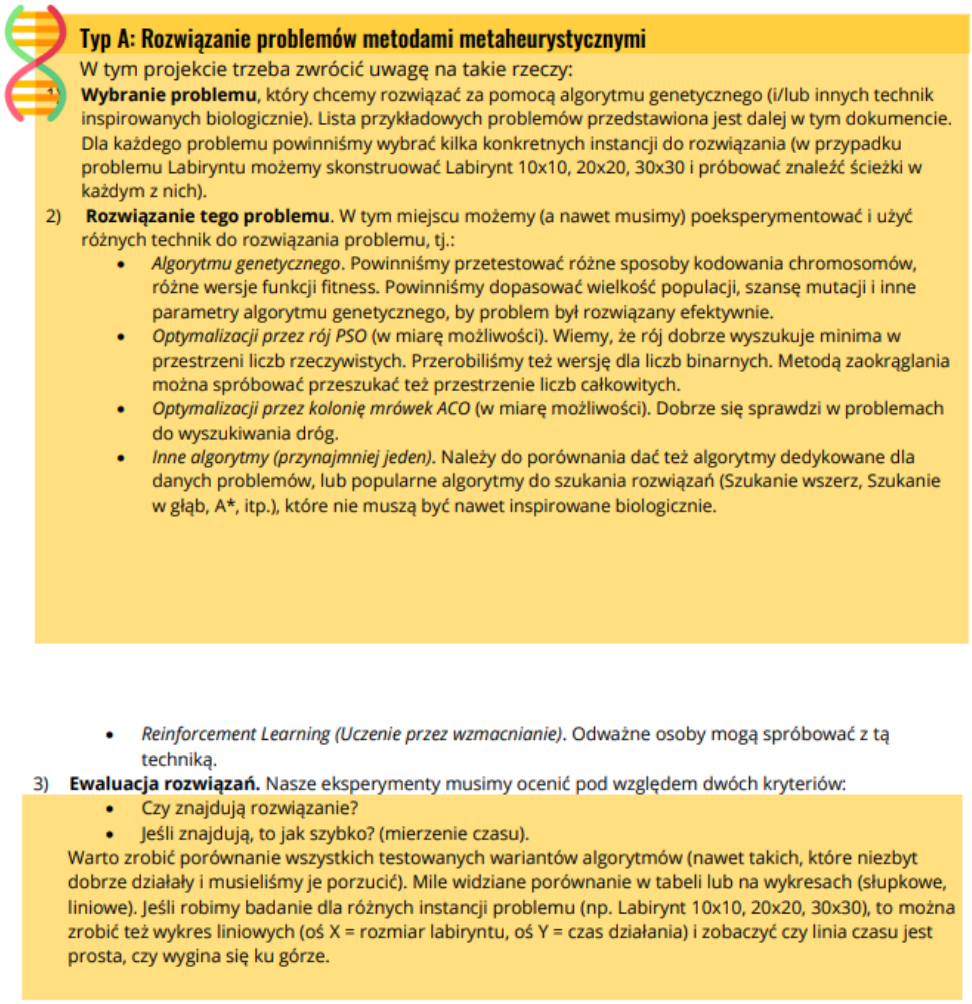
Jakub Zabłocki

styczeń 2023

1 Opis

Nonogramy to popularna gra logiczna, w której celem jest odgadnięcie ukrytego obrazka na podstawie dostępnych informacji o rzędach i kolumnach. W projekcie przedstawiono rozwiązanie nonogramów przy użyciu algorytmu genetycznego oraz algorytmu PSO (Particle Swarm Optimization). Program został napisany w języku python, a wyniki działania algorytmów zostały porównane pod względem różnych funkcji fitness, czasów działania oraz procentowych wyników. Celem projektu jest zaprezentowanie skuteczności obu algorytmów.

2 Typ i treść zadania



Typ A: Rozwiązanie problemów metodami metaheurystycznymi

W tym projekcie trzeba zwrócić uwagę na takie rzeczy:

1) **Wybranie problemu**, który chcemy rozwiązać za pomocą algorytmu genetycznego (i/lub innych technik inspirowanych biologicznie). Lista przykładowych problemów przedstawiona jest dalej w tym dokumencie. Dla każdego problemu powinniśmy wybrać kilka konkretnych instancji do rozwiązania (w przypadku problemu Labiryntu możemy skonstruować Labirynt 10x10, 20x20, 30x30 i próbować znaleźć ścieżki w każdym z nich).
2) **Rozwiązanie tego problemu**. W tym miejscu możemy (a nawet musimy) poeksperymentować i użyć różnych technik do rozwiązania problemu, tj.:
   - *Algorytmu genetycznego*. Powinniśmy przetestować różne sposoby kodowania chromosomów, różne wersje funkcji fitness. Powinniśmy dopasować wielkość populacji, szansę mutacji i inne parametry algorytmu genetycznego, by problem był rozwiązany efektywnie.
   - *Optymalizacji przez rój PSO* (w miarę możliwości). Wiemy, że rój dobrze wyszukuje minima w przestrzeni liczb rzeczywistych. Przerobiliśmy też wersję dla liczb binarnych. Metodą zaokrąglania można spróbować przeszukać też przestrzenie liczb całkowitych.
   - *Optymalizacji przez kolonię mrówek ACO* (w miarę możliwości). Dobrze się sprawdzi w problemach do wyszukiwania dróg.
   - *Inne algorytmy (przynajmniej jeden)*. Należy do porównania dać też algorytmy dedykowane dla danych problemów, lub popularne algorytmy do szukania rozwiązań (Szukanie wszerz, Szukanie w głąb, A*, itp.), które nie muszą być nawet inspirowane biologicznie.
   - *Reinforcement Learning (Uczenie przez wzmacnianie)*. Odważne osoby mogą spróbować z tą techniką.
3) **Ewaluacja rozwiązań.** Nasze eksperymenty musimy ocenić pod względem dwóch kryteriów:
   - Czy znajdują rozwiązanie?
   - Jeśli znajdują, to jak szybko? (mierzenie czasu).

Warto zrobić porównanie wszystkich testowanych wariantów algorytmów (nawet takich, które niezbyt dobrze działały i musieliśmy je porzucić). Mile widziane porównanie w tabeli lub na wykresach (słupkowe, liniowe). Jeśli robimy badanie dla różnych instancji problemu (np. Labirynt 10x10, 20x20, 30x30), to można zrobić też wykres liniowych (oś X = rozmiar labiryntu, oś Y = czas działania) i zobaczyć czy linia czasu jest prosta, czy wygina się ku górze.

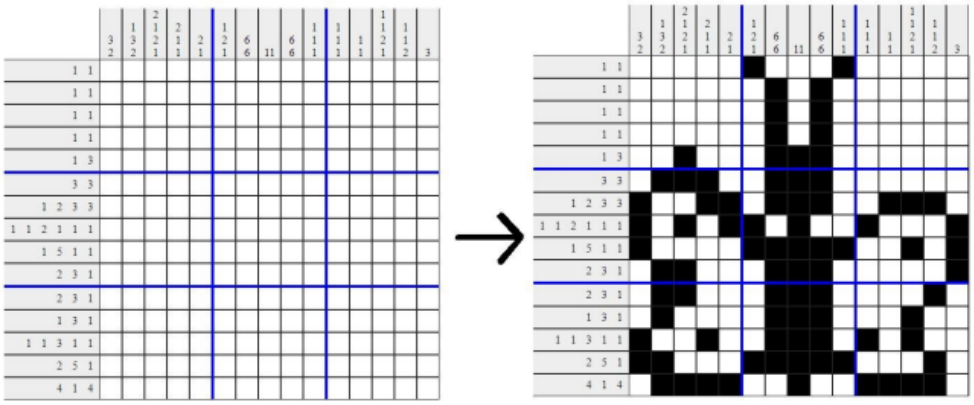
Rysunek 1: typ zadania

Temat 2 typ A

Rozwiązywanie nonogramów

Nonogram (zwany też obrazkiem logicznym) 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. Przykładowo, jeśli przy wierszu stoi "2 3 1", to znaczy, że w danym wierszu jest ciąg dwóch zamalowanych kratek, przerwa (przynajmniej jedna biała kratka), trzy zamalowane kratki, przerwa, jedna zamalowana kratka. Umiejscowienie zamalowanych ciągów nie jest wskazane.

Mając daną nieuzupełnioną planszę, pytamy: czy istnieje rozwiązanie dla tej zagadki (tzn. odpowiednie zamalowanie pól bez żadnych sprzeczności)? Poniżej przedstawiono instancję problemu (po lewej) i jej rozwiązanie (po prawej). Rozważ, jak znajdować rozwiązanie za pomocą algorytmu genetycznego, PSO lub innych algorytmów.



Rysunek 2: treść zadania

3 Algorytmy

3.1 Algorytm genetyczny

Algorytm genetyczny jest metodą ewolucyjną, która polega na symulowaniu procesu dziedziczenia i mutacji genów w celu znalezienia optymalnego rozwiązania problemu. W przypadku rozwiązywania nonogramów, algorytm genetyczny jest stosowany do generowania populacji początkowej rozwiązań, które są następnie poddawane operacjom krzyżowania i mutacji w celu stworzenia nowych rozwiązań. Populacja jest poddawana kolejnym pokoleniom, a najlepsze rozwiązania są zachowywane do kolejnego etapu. Proces ten jest powtarzany do momentu znalezienia rozwiązania, które spełnia wymagania problemu.

3.2 Algorytm PSO

Algorytm PSO (Particle Swarm Optimization) jest metodą optymalizacji, która polega na symulowaniu ruchu cząsteczek w przestrzeni poszukiwań. W przypadku rozwiązywania nonogramów, algorytm PSO jest stosowany do generowania populacji początkowej rozwiązań, które są następnie poddawane operacjom ruchu i aktualizacji prędkości w celu znalezienia optymalnego rozwiązania. Populacja jest poddawana kolejnym iteracjom, a najlepsze rozwiązania są zachowywane do kolejnego etapu. Proces ten jest powtarzany do momentu znalezienia rozwiązania, które spełnia wymagania problemu.

Oba algorytmy są używane do rozwiązywania nonogramów, jednak różnią się one w sposobie generowania i aktualizacji rozwiązań, co może prowadzić do różnych czasów działania oraz skuteczności w rozwiązywaniu problemu.

4 Implementacja

4.1 Użyte technologie

4.1.1 PyCharm

PyCharm to popularne środowisko programistyczne (IDE) dla języka Python, oferujące szereg narzędzi do pomocy programistom w tworzeniu, testowaniu i debugowaniu kodu. Oferuje ono funkcje takie jak automatyczne uzupełnianie kodu, analiza składni, rozbudowany system debugowania, integrację z systemami kontroli wersji i wiele innych. PyCharm jest przeznaczony dla programistów zarówno początkujących, jak i zaawansowanych i jest jednym z najpopularniejszych środowisk programistycznych dla języka Python.

4.1.2 Python

Python to wysokiego poziomu, uniwersalny język programowania, który jest popularny zarówno w nauce, jak i przemyśle. Jest on łatwy do nauki dla początkujących, ale jednocześnie posiada rozbudowane funkcjonalności dla zaawansowanych programistów. Python jest często wykorzystywany do tworzenia skryptów, aplikacji internetowych, a także w naukach przyrodniczych, statystyce i uczeniu maszynowym.

4.1.3 PyGAD

PyGAD to pakiet open-source do algorytmów genetycznych dla języka Python. Zawiera on implementację standardowych operacji genetycznych, takich jak selekcja, krzyżowanie i mutacja, a także dodatkowe funkcje, takie jak wielowątkowość i automatyczne dopasowywanie parametrów. Ten pakiet jest skierowany do początkujących programistów, którzy chcą wykorzystać algorytm genetyczny do rozwiązywania różnych problemów.

4.1.4 Pyswarms

Pyswarms to pakiet open-source do algorytmu PSO dla języka Python. Zawiera on implementację standardowych operacji PSO, takich jak aktualizacja prędkości i pozycji cząsteczek oraz

funkcje dodatkowe, takie jak wielowątkowość i automatyczne dopasowywanie parametrów. Dzięki pakietowi pyswarms, programiści mogą łatwo implementować algorytm PSO w swoich projektach i dostosować go do swoich potrzeb za pomocą dostępnych parametrów.

4.2 Struktura programu

Struktura programu składa się z 4 osobnych plików:

4.2.1 algorytm_genetyczny.py

Ten plik zawiera klasę Algorytm_genetyczny, która jest odpowiedzialna za implementację algorytmu genetycznego w projekcie. Klasa ta posiada funkcję uruchom, która inicjuje proces rozwiązywania nonogramu przy użyciu algorytmu genetycznego. Funkcja ta wykorzystuje trzy różne funkcje fitness, które służą do oceny jakości rozwiązania.

4.2.2 algorytm_PSO.py

Plik zawiera klasę Algorytm_PSO, która jest odpowiedzialna za implementację algorytmu PSO w projekcie. Klasa ta posiada funkcję uruchom, która inicjuje proces rozwiązywania nonogramu przy użyciu algorytmu PSO. Funkcja ta wykorzystuje trzy różne funkcje fitness, które służą do oceny jakości rozwiązania.

4.2.3 main.py

Główny plik programu, który odpowiada za scalenie całego projektu w całość i uruchomienie go. Plik ten używa klas z pliku algorytm_genetyczny.py i algorytm_PSO.py do rozwiązywania nonogramu i generowanie statystyk na temat działania algorytmów.

4.2.4 nonogramy.py

Plik ten zawiera klasę Nonogram, która jest odpowiedzialna za tworzenie obiektów nonogramów o różnych rozmiarach. Jest to baza danych w tym projekcie.

4.3 Implementacja algorytmów

4.3.1 Genetyczny

Odpowiedni plik zawiera klasę Algorytm_genetyczny, która implementuje algorytm genetyczny w projekcie. Klasa ta posiada główną metodę "uruchom", która jest odpowiedzialna za rozwiązanie nonogramu przy użyciu algorytmu genetycznego.

```
class Algorytm_genetyczny:
    def uruchom(self, nonogram, fitness_nazwa):
```

Rysunek 3: algorytm_genetyczny.py

Metoda "uruchom" przyjmuje dwa argumenty: "nonogram" - obiekt nonogramu klasy Nonogram, który ma zostać rozwiązany oraz "fitness_nazwa" - nazwę jednej z trzech funkcji fitness, która ma być użyta do oceny jakości rozwiązania. Funkcja fitness jest odpowiedzialna za ocenę jakości rozwiązania na podstawie stopnia zgodności z danymi wejściowymi.

4.3.2 Rój cząsteczek

Odpowiedni plik zawiera klasę Algorytm_PSO, która implementuje algorytm PSO w projekcie. Klasa ta posiada główną metodę "uruchom", która jest odpowiedzialna za rozwiązanie nonogramu przy użyciu algorytmu PSO.

```
class Algorytm_PSO:
    def uruchom(self, nonogram, fitness_nazwa, generations):
```

Rysunek 4: algorytm_PSO.py

Metoda "uruchom" przyjmuje trzy argumenty: "nonogram" - obiekt nonogramu, który ma zostać rozwiązany, "fitness_nazwa" - nazwę jednej z trzech funkcji fitness, która ma być użyta do oceny jakości rozwiązania oraz "generations" - liczbę generacji, które mają zostać przeprowadzone przez algorytm PSO.

4.4 Baza danych

Plik nonogramy.py jest bazą danych projektu i zawiera klasę Nonogram, która jest odpowiedzialna za tworzenie obiektów nonogramów. W tym pliku, są tworzone nonogramy o różnych rozmiarach od 3x3 do 15x15. Każdy obiekt nonogramu posiada trzy pola: "columns", "rows" i "expected_solution".

```
1  class Nonogram:
2      def __init__(self, columns, rows, expected_solution):
3          self.columns = columns
4          self.rows = rows
5          self.expected_solution = expected_solution
```

Rysunek 5: klasa Nonogram

"Columns" i "rows" zawierają informacje o ilości pól do wypełnienia w kolumnach i wierszach, a "expected_solution" zawiera oczekiwaną odpowiedź dla danego nonogramu. Obiekty nonogramów są używane w pliku main.py przez algorytm genetyczny i PSO w celu rozwiązania nonogramów. Przykład tworzenia nonogramu 4x4:

```
nonogram_4x4_1 = Nonogram([[1], [2], [3], [1]],
                          [[2], [1], [3], [1]],
                          [[0, 0, 1, 1], [0, 0, 1, 0], [1, 1, 1, 0], [0, 1, 0, 0]])
```

Rysunek 6: tworzenie nonogramu

4.5 Funkcja fitness

Inaczej funkcja przystosowania jest ważnym elementem algorytmów genetycznego i PSO- jej celem jest ocena jakości przystosowania osobnika. Przystosowanie to związane jest z jakością danego rozwiązania. W projekcie ta funkcja została zaimplementowana na trzy różne sposoby, które coraz bardziej uwzględniają różne czynniki, dzięki czemu ich wydajność jest coraz lepsza.

Każda z funkcji przyjmuje jako rozwiązanie listę liczb o długości odpowiadającej ilości pól w nonogramie (np. nonogram 3x3 ma długość 9). Lista ta składa się z liczb o wartościach 0 lub 1, gdzie 0 oznacza pole niezamalowane, a 1 oznacza pole zamalowane.

Funkcje fitness dla algorytmu genetycznego oraz PSO różnią się tym, że na końcu funkcji dla PSO rozwiązanie zostaje zapisane w specyficzny sposób:

```
nonlocal best_fitness, best_solution

if rozwiazanie > best_fitness:
    best_fitness = rozwiazanie
    best_solution = solution
return -rozwiazanie
```

Rysunek 7: zapis rozwiązania w PSO

```
best_fitness = float('-inf')
best_solution = None
```

Rysunek 8: zmienne

4.5.1 Prosta wersja fitness

Ta wersja fitness jest jednym z podstawowych i najprostszych wariantów funkcji fitness. Funkcja ta jest rygorystyczna, oznacza to, że iterując przez kolumny i wiersze gdy wystąpi jedna niezgodność pomiędzy oczekiwaną odpowiedzią a rozwiązaniem, cała kolumna/wiersz automatycznie otrzymuje ujemny punkt.

Pierwsze co robi funkcja to zmienna rozwiązanie jest ustawiona na 0, a następnie rozwiązanie jest dzielone na wiersze. Następnie tworzy listę "wynik_rows", w której każdy element jest grupą jedynek z danego wiersza. To jest osiągane przez iterację po wierszach rozwiązania i zastosowanie funkcji groupby, która dzieli ciąg na grupy z takim samym elementem.

Następnie tworzy listę "wynik_columns", w której każdy element jest grupą jedynek z danej kolumny. To jest osiągane przez iterację po kolumnach rozwiązania, za pomocą funkcji column_extractor która zwraca elementy z danej kolumny, a następnie zastosowanie funkcji groupby, która dzieli ciąg na grupy z takim samym elementem.

Następnie, funkcja sprawdza, czy wynik_rows i wynik_columns są równe odpowiedniemu wierszowi lub kolumnie z danymi wejściowymi (rows i cols). Jeśli nie są równe, rozwiązanie jest dekrementowane o 1. Proces ten jest powtarzany dla każdego wiersza i kolumny.

Na końcu, funkcja zwraca rozwiązanie jako wynik, co jest miarą jakości rozwiązania. Im większa wartość rozwiązania, tym lepsze rozwiązanie.

```python
def column_extractor(matrix, column_index):
    return [row[column_index] for row in matrix]

# definiujemy funkcję fitness
def fitness_1(solution, solution_idx):
    rozwiazanie = 0
    solution = [solution[i:i + len(rows)] for i in range(0, len(solution), len(rows))]
    wynik_rows = []
    # lista w ktorej kazdy element jest grupa jedynek z danego wiersza
    for row in solution:
        wynik_rows.append([len(group) for group in groupby(row) if group[0] == 1])

    wynik_columns = []
    # lista w ktorej kazdy element jest grupa jedynek z danej kolumny
    for column in range(len(solution[0])):
        wynik_columns.append(
            [len(list(group)) for val, group in groupby(column_extractor(solution, column)) if val == 1])

    for i in range(len(rows)):
        if rows[i] != wynik_rows[i]:
            rozwiazanie -= 1
        if cols[i] != wynik_columns[i]:
            rozwiazanie -= 1
    return rozwiazanie
```

Rysunek 9: funkcja fitness_1

4.5.2 Lepsza wersja fitness

Jest to ulepszona wersja funkcji fitness, która zamiast traktowania wyniku osobnika bardzo rygorystycznie, posiada ona większą ilość kryterii.

Początek tej funkcji jest bardzo podobny do poprzedniej, jednak jest zapisany w inny sposób, który jest bardziej zwięzły, skrócony i przy użyciu list comprehension jest bardziej czytelny. Kod został tak zmodyfikowany, aby nie potrzebna była funkcja column_extractor.

Kryteria porównawcze wynik_rows i wynik_columns z rows i cols zostały rozszerzone o:

- Jeśli długości kolumn/wierszy są różne, rozwiązanie jest dekrementowane o różnicę tych długości,
- Porównuje każdy element wiersza rows i wynik_rows oraz każdy element kolumny cols i wynik_columns, i jeśli znajdzie jakieś różnice to odejmuje punkty od rozwiązania.

```
def fitness_2(solution, solution_idx):
    rozwiazanie = 0
    solution = [solution[i:i + len(rows)] for i in range(0, len(solution), len(rows))]

    wynik_rows = [[len(list(group)) for value, group in groupby(i) if value == 1] for i in solution]

    wynik_columns = [[len(list(group)) for value, group in groupby(i) if value == 1] for i in zip(*solution)]

    for i in range(len(rows)):
        rozwiazanie -= abs(len(rows[i]) - len(wynik_rows[i]))
        rozwiazanie -= abs(len(cols[i]) - len(wynik_columns[i]))
        for j in range(min(len(rows[i]), len(wynik_rows[i]))):
            if rows[i][j] != wynik_rows[i][j]:
                rozwiazanie -= 1
        for j in range(min(len(cols[i]), len(wynik_columns[i]))):
            if cols[i][j] != wynik_columns[i][j]:
                rozwiazanie -= 1

    return rozwiazanie
```

Rysunek 10: funkcja fitness_2

4.5.3 Najlepsza wersja fitness

Jest to ostatnia, finalna wersja funkcji fitness w projekcie. Została ona stworzona poprzez dodanie do funkcji fitness_2 kolejnych kryteriów, które bazują na expected_solution, czyli na wyniku, który powinniśmy docelowo uzyskać. Porównując solution z expected_solution, uzyskujemy statystyki. W tej funkcji również dodano nagrodę za dobrze wypełnione pole.

Funkcja fitness_3 różni się od fitness_2 tymi kryteriami:

- Porównuje każdy z elementów solution z expected_solution
- Odejmuje punkty za niewypełnienie pola, które powinno byc
- Dodaje punkty za prawidłowo wypełnione 1

- Porównuje, czy dane pole jest odpowiednio wypełnione zgodnie z expected_solution, biorąc pod uwagę stan sąsiednich pól. Jeśli pole jest odpowiednio wypełnione, punkt jest dodawany, w przeciwnym razie odejmowany.

```python
def fitness_3(solution, solution_idx):
    rozwiazanie = 0
    solution = [solution[i:i + len(rows)] for i in range(0, len(solution), len(rows))]

    wynik_rows = [[len(list(group)) for value, group in groupby(i) if value == 1] for i in solution]

    wynik_columns = [[len(list(group)) for value, group in groupby(i) if value == 1] for i in zip(*solution)]

    for i in range(len(rows)):
        rozwiazanie -= abs(len(rows[i]) - len(wynik_rows[i]))
        rozwiazanie -= abs(len(cols[i]) - len(wynik_columns[i]))
        for j in range(min(len(rows[i]), len(wynik_rows[i]))):
            if rows[i][j] != wynik_rows[i][j]:
                rozwiazanie -= 1
        for j in range(min(len(cols[i]), len(wynik_columns[i]))):
            if cols[i][j] != wynik_columns[i][j]:
                rozwiazanie -= 1

    for i in range(len(solution)):
        for j in range(len(solution[i])):
            if solution[i][j] != expected_solution[i][j]:
                rozwiazanie -= 2
            elif solution[i][j] == 0 and expected_solution[i][j] == 1:
                rozwiazanie -= 1
            elif solution[i][j] == 1 and expected_solution[i][j] == 1:
                rozwiazanie += 1
            if i > 0 and j > 0:
                if solution[i][j] == 1 and expected_solution[i][j] == 1 and expected_solution[i][j - 1] == 0 and expected_solution[i - 1][j] == 0:
                    rozwiazanie += 1
                elif solution[i][j] != expected_solution[i][j] and expected_solution[i][j - 1] == 0 and expected_solution[i - 1][j] == 0:
                    rozwiazanie -= 1
            if i < len(solution) - 1 and j < len(solution[i]) - 1:
                if solution[i][j] == 1 and expected_solution[i][j] == 1 and expected_solution[i + 1][j] == 0 and expected_solution[i][j + 1] == 0:
                    rozwiazanie += 1
                elif solution[i][j] != expected_solution[i][j] and expected_solution[i + 1][j] == 0 and expected_solution[i][j + 1] == 0:
                    rozwiazanie -= 1


    return rozwiazanie
```

Rysunek 11: funkcja fitness_3

5 Parametry algorytmów

Każdy z algorytmów posiada różne zmienne, które trzeba przypisać.

5.1 Algorytm genetyczny

```python
gene_space = [0, 1]
sol_per_pop = 100
num_genes = len(rows) * len(cols)
num_parents_mating = 50
num_generations = 100
keep_parents = 2
parent_selection_type = "sss"
crossover_type = "single_point"
mutation_type = "random"
mutation_percent_genes = 12
```

gene_space - jest to lista wartości, które mogą być przyjmowane przez geny w algorytmie. W tym przypadku jest to 0 lub 1, co odpowiada niezamalowanej/zamalowanej komórce w nonogramie,

sol_per_pop - oznacza liczbę rozwiązań w każdej populacji. W tym przypadku jest to 100,

num_genes - jest to liczba genów w każdym rozwiązaniu. Jest to rozmiar nonogramu,

num_parents_mating - oznacza liczbę rodziców, którzy będą się krzyżować w celu utworzenia nowej populacji. Tutaj jest to 50,

num_generations - oznacza liczbę pokoleń, które będą generowane przez algorytm. Zostało to ustawione na 100,

keep_parents - oznacza liczbę rodziców, którzy będą przechowywani w nowej populacji,

parent_selection_type - oznacza typ selekcji rodziców, który będzie stosowany w algorytmie,

crossover_type - określa metodę krzyżowania,

mutation_type - rodzaj mutacji,

mutation_percent_genes - procent mutacji.

5.2 Algorytm PSO

```
particles = len(rows) * len(cols)
num_generations = generations
options = {'c1': 1.6, 'c2': 0.8, 'w': 0.8, 'k': 2, 'p': 2}
```

particles - jest to liczba cząsteczek (osobników). Jest to rozmiar nonogramu

num_generations - oznacza liczbę iteracji, czyli liczbę razy jaką algorytm ma się powtarzać w celu osiągniecia jak najlepszego rozwiązania. zmienna "generations" jest tutaj przekazywana podczas uruchamiania algorytmu

```
class Algorytm_PSO:
    def uruchom(self, nonogram, fitness_nazwa, generations):
```

Rysunek 12: funkcja uruchom()

c1 - parametr konfiguracyjny odpowiedzialny za wpływ najlepszego dotychczasowego rozwiązania cząsteczki

c2 - zmienna odpowiedzialna za wpływ najlepszego rozwiązania całego roju.

w - kontroluje wpływ poprzedniej pozycji cząsteczki na aktualną pozycję.

k - liczba sąsiadów, z którą działa pojedyncza jednostka

p - opcja 1- wartość bezwzględna, 2- odległość euklidesowa

6 Wyniki i statystyka

Wyniki kompilacji programu dla obu algorytmów używając różnych nonogramów, różnych funkcji fitness oraz różnych zmiennych.

Funkcje, dzięki którym otrzymamy statystyki znajdują się w pliku main.py:

```python
import nonogramy
import algorytm_genetyczny
import algorytm_PSO


genetyczny = algorytm_genetyczny.Algorytm_genetyczny()
pso = algorytm_PSO.Algorytm_PSO()

nonogramy = nonogramy.nonogramy
fitness = ["fitness_1", "fitness_2", "fitness_3"]


def statystyki_generyczny(nonogram, fitness, ilosc_prob):
    statystyki = []
    for i in range(ilosc_prob):
        statystyki.append(genetyczny.uruchom(nonogram, fitness))
    #print(statystyki)
    sredni_czas = 0
    sredni_wynik = 0
    ukonczony = 0
    for i, j, k in statystyki:
        sredni_czas += i
        sredni_wynik += j
        ukonczony += k
    return "sredni czas: " + str(round(sredni_czas / len(statystyki), 2)), \
           "sredni wynik: " + str(round(sredni_wynik / len(statystyki), 2)), \
           "ilosc ukonczonych: " + str(ukonczony)


def statystyki_pso(nonogram, fitness, ilosc_generacji, ilosc_prob):
    statystyki = []
    for i in range(ilosc_prob):
        statystyki.append(pso.uruchom(nonogram, fitness, ilosc_generacji))
    #print(statystyki)
    sredni_czas = 0
    sredni_wynik = 0
    ukonczony = 0
    for i, j, k in statystyki:
        sredni_czas += i
        sredni_wynik += j
        ukonczony += k
    return "sredni czas: " + str(round(sredni_czas / len(statystyki), 2)), \
           "sredni wynik: " + str(round(sredni_wynik / len(statystyki), 2)), \
           "ilosc ukonczonych: " + str(ukonczony)
```

Rysunek 13: plik do statystyk

Przykładowy wynik kompilacji:

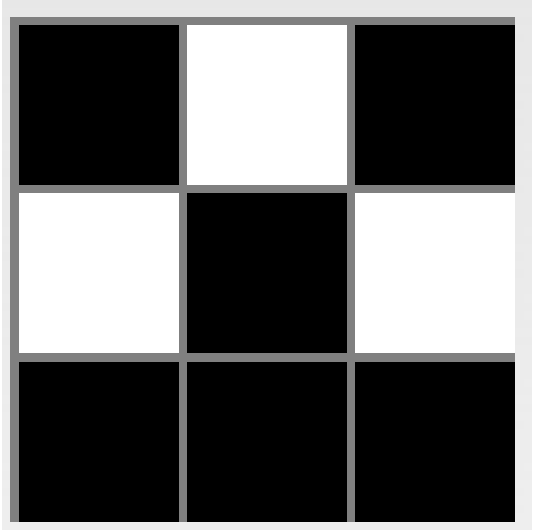
```
wynik rozwiązywania nonogramu przy użyciu algorytmu genetycznego:
Udało się prawidłowo rozwiązać nonogram.
time: 0.286
Best solution: [[1, 0, 1], [0, 1, 0], [1, 1, 1]]
Best fitness: 0

(0.286, 100, 1)

Process finished with exit code 0
```

Rysunek 14: wynik kompilacji nonogramu 3x3, fitness_1

6.1 Nonogram 3x3

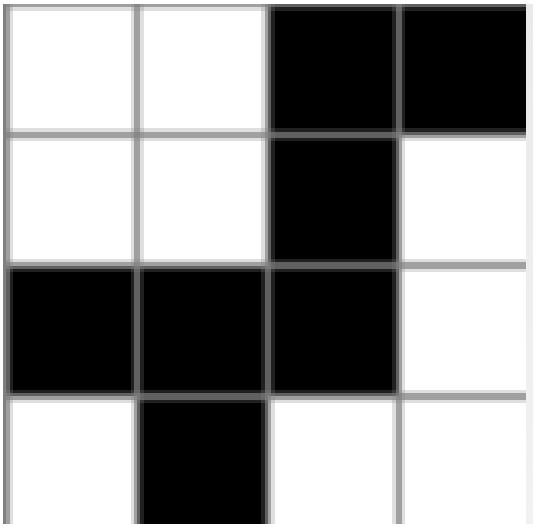


Rysunek 15: nonogram 3x3

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	3x3	genetyczny	0.28	87,78	9	20		
fitness_1	3x3	pso	0.38	77.22	8	20		2500
fitness_2	3x3	genetyczny	0.31	100	20	20		
fitness_2	3x3	pso	0.38	95.56	19	20		2500
fitness_3	3x3	genetyczny	0.42	100	20	20		
fitness_3	3x3	pso	0.41	100	20	20		2500

Rysunek 16: statystyki 3x3

6.2 Nonogram 4x4

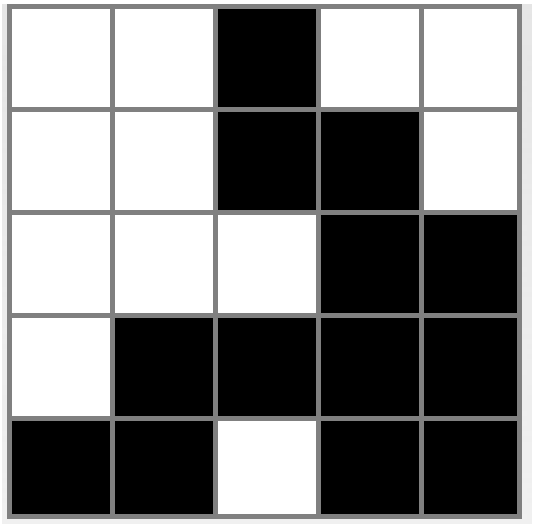


Rysunek 17: nonogram 4x4

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	4x4	genetyczny	0.32	63.75	0	20		
fitness_1	4x4	pso	0.4	58.75	0	20		2500
fitness_2	4x4	genetyczny	0.35	83.12	14	20		
fitness_2	4x4	pso	0.4	63.75	1	20		2500
fitness_3	4x4	genetyczny	0.53	100	20	20		
fitness_3	4x4	pso	0.43	100	20	20		2500

Rysunek 18: statystyki 4x4

6.3 Nonogram 5x5

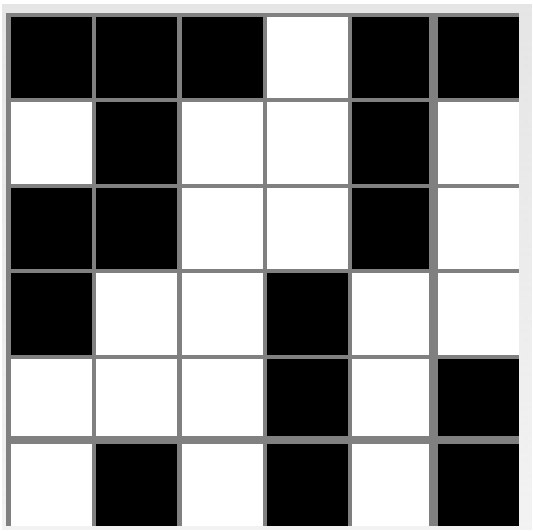


Rysunek 19: nonogram 5x5

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	5x5	genetyczny	0.38	70	2	20		
fitness_1	5x5	pso	0.43	61.4	0	20		2500
fitness_2	5x5	genetyczny	0.42	92.2	17	20		
fitness_2	5x5	pso	0.43	56.2	1	20		2500
fitness_3	5x5	genetyczny	0.71	100	20	20		
fitness_3	5x5	pso	0.49	99.8	19	20		2500

Rysunek 20: statystyki 5x5

6.4 Nonogram 6x6

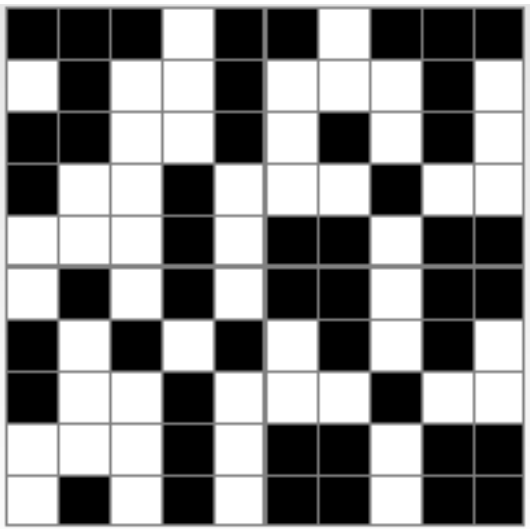


Rysunek 21: nonogram 6x6

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	6x6	genetyczny	0.45	56.53	0	20		
fitness_1	6x6	pso	0.45	54.72	0	20		2500
fitness_2	6x6	genetyczny	0.5	97.36	14	20		
fitness_2	6x6	pso	0.47	60	0	20		2500
fitness_3	6x6	genetyczny	0.83	100	20	20		
fitness_3	6x6	pso	0.53	99.86	19	20		2500

Rysunek 22: statystyki 6x6

6.5 Nonogram 10x10

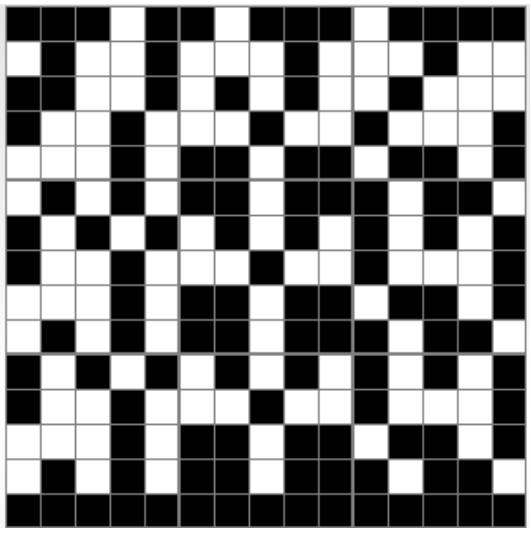


Rysunek 23: nonogram 10x10

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	10x10	genetyczny	0.84	51.2	0	50		
fitness_1	10x10	pso	0.64	50.3	0	50		2500
fitness_2	10x10	genetyczny	0.89	52.23	0	50		
fitness_2	10x10	pso	0.65	51	0	50		2500
fitness_3	10x10	genetyczny	2.0	87.8	0	50		
fitness_3	10x10	pso	0.84	88.9	0	50		2500

Rysunek 24: statystyki 10x10

6.6 Nonogram 15x15



Rysunek 25: nonogram 15x15

Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_1	15x15	genetyczny	1.43	48.73	0	20		
fitness_1	15x15	pso	1.05	48.76	0	20		2500
fitness_2	15x15	genetyczny	1.53	48.64	0	20		
fitness_2	15x15	pso	1.12	50.07	0	20		2500
fitness_3	15x15	genetyczny	4.15	76.11	0	20		
fitness_3	15x15	pso	1.51	76.93	0	20		2500

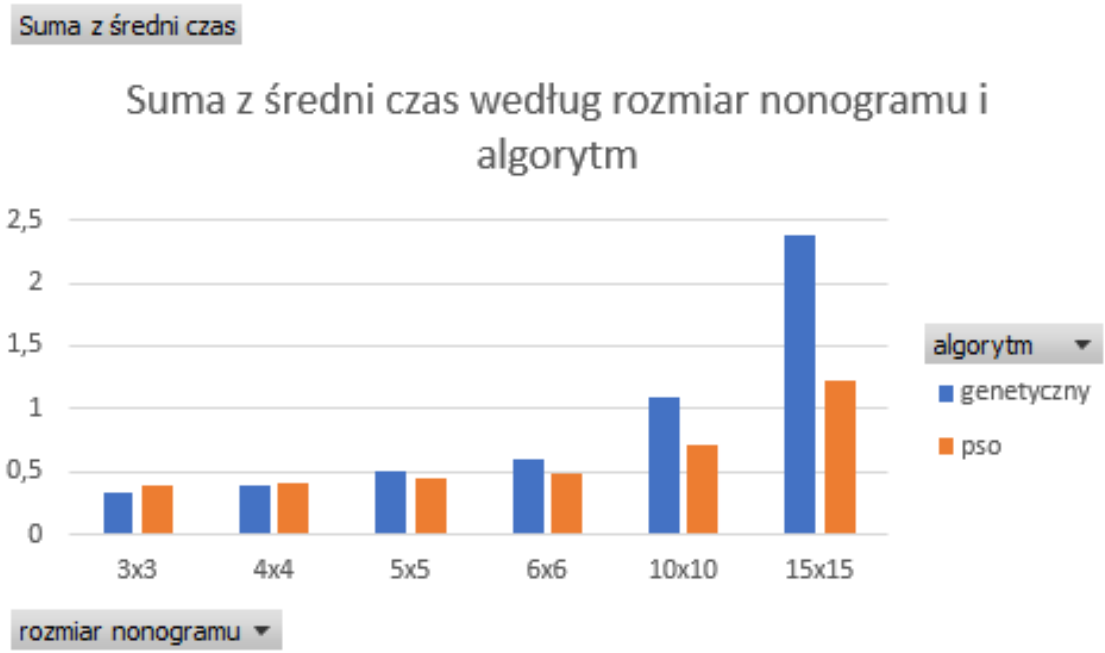
Rysunek 26: statystyki 15x15

Dla samego PSO:

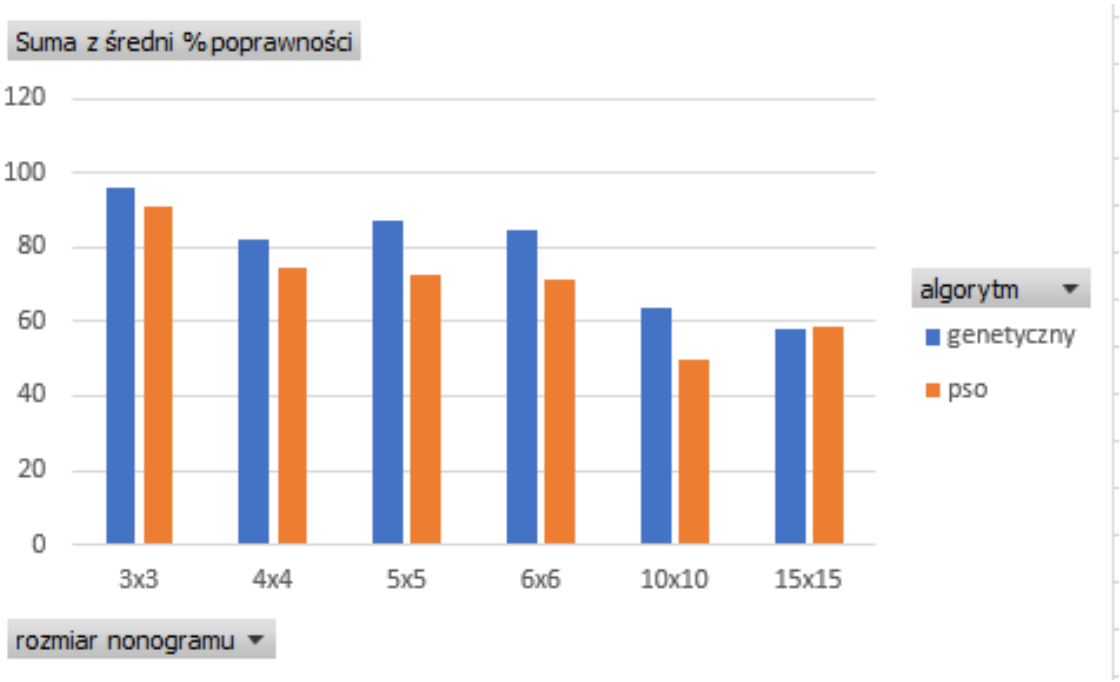
Funkcja fitness:	rozmiar nonogramu	algorytm	średni czas	średni % poprawności	ilość ukończonych	ilość prób		generacje
fitness_3	15x15	pso	6.39	92.27	0	5		10000
fitness_3	15x15	pso	12.59	96.44	0	5		20000
fitness_3	15x15	pso	18.44	98.49	0	5		30000
fitness_3	15x15	pso	25.27	99.02	1	5		40000
fitness_3	15x15	pso	31.41	99.2	2	5		50000

Rysunek 27: drugie statystyki 15x15

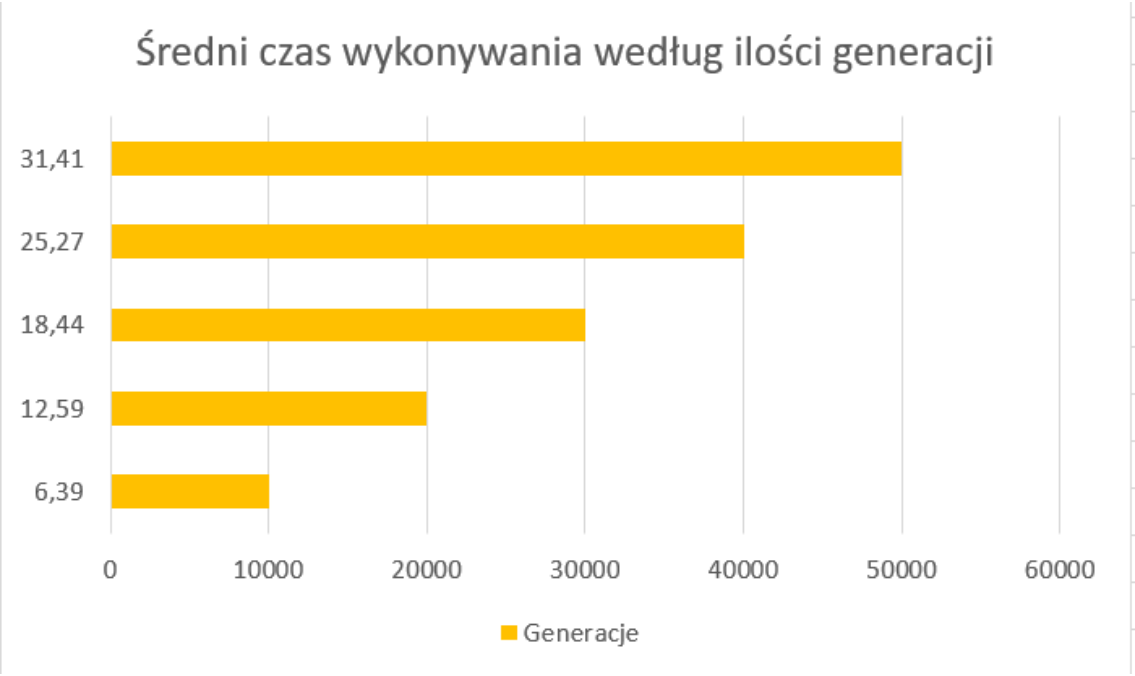
6.7 Wykresy



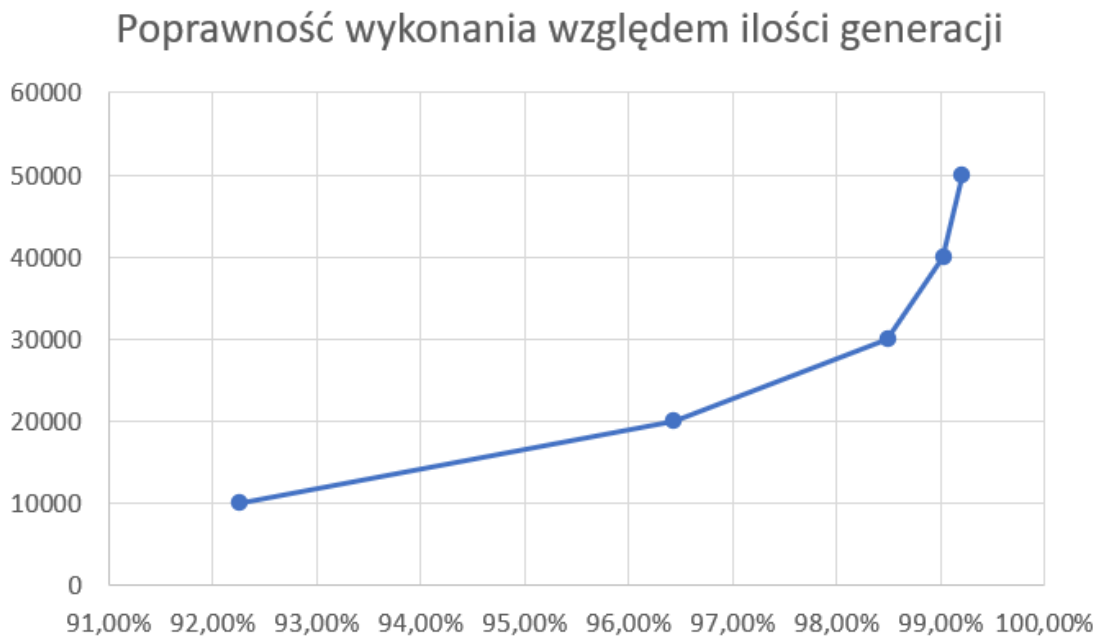
Rysunek 28: Średni czas wykonania



Rysunek 29: Średnia poprawność



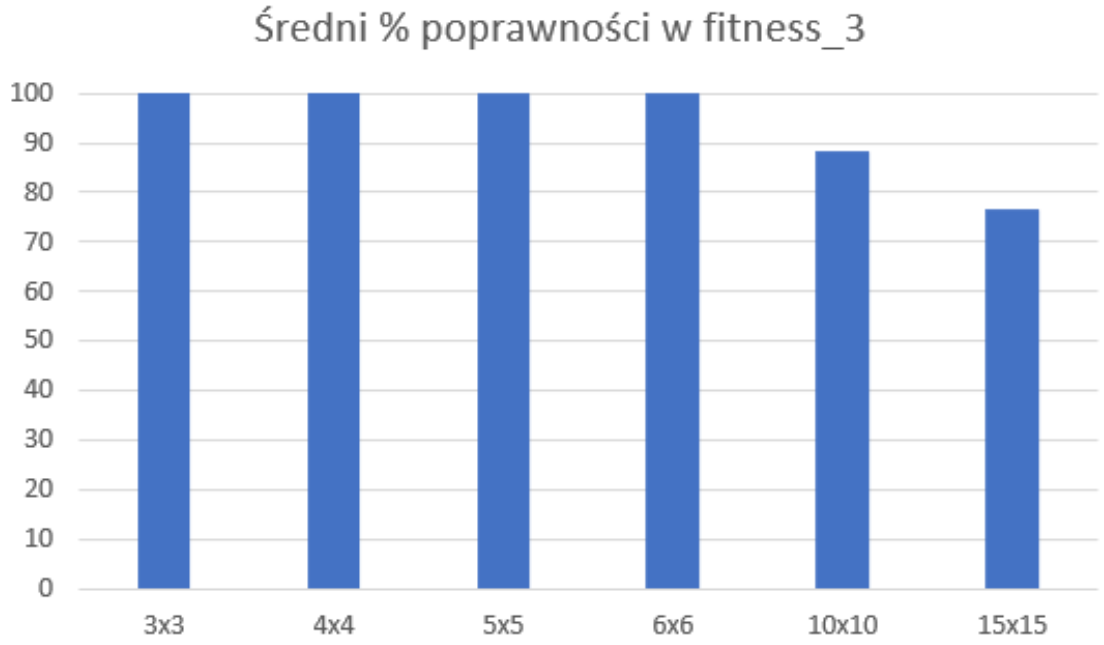
Rysunek 30: Średni czas dla PSO, nonogram 15x15



Rysunek 31: Średnia poprawność dla PSO, nonogram 15x15

Rysunek 32: Poprawność w fitness_1

Rysunek 33: Poprawność w fitness_2



Rysunek 34: Poprawność w fitness_3

7 Podsumowanie

Projekt dotyczył rozwiązywania nonogramów za pomocą dwóch różnych algorytmów: algorytmu genetycznego oraz algorytmu PSO. W projekcie zaimplementowano trzy różne funkcje fitness, które różniły się poziomem rygorystyczności i stopniem skomplikowania.

Wyniki testów wykazały, że algorytm genetyczny jest trochę lepszy dla mniejszych nonogramów, jednak im większy nonogram tym algorytm PSO nadrabia różnicę.

Różnica między funkcjami fitness była ogromna i zauważalna po przedstawieniu wykresów, co potwierdza, że dobór odpowiedniej funkcji fitness jest kluczowy dla osiągnięcia najlepszych wyników.

Bibliografia

[1] https://pygad.readthedocs.io/en/latest

[2] https://pygad.readthedocs.io/en/latest