# Inteligencja obliczeniowa - Projekt 1

Mateusz Stapaj

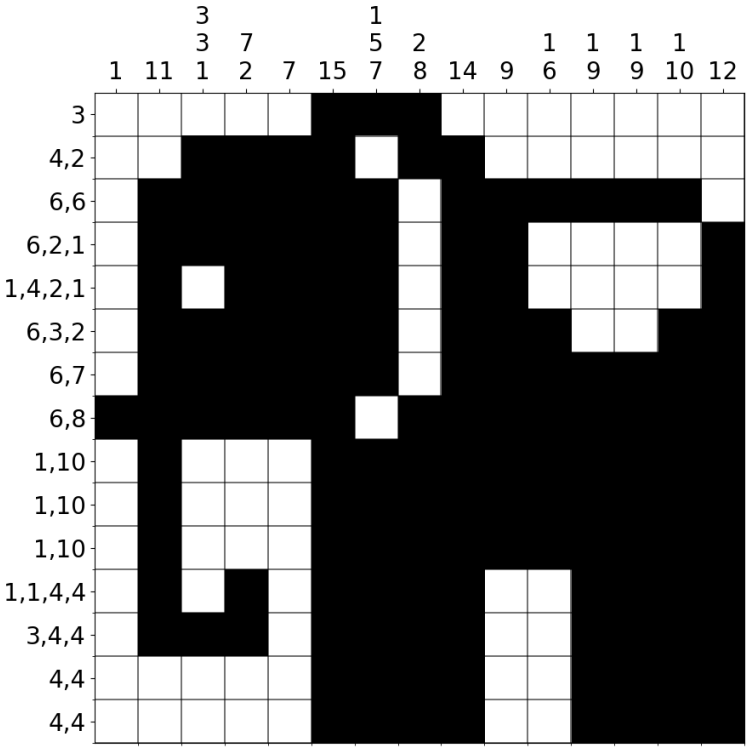
21 kwietnia 2022

## 1 Wstęp

Projekt polega na rozwiązanie zagadki "Nonogram" za pomocą algorytmów genetycznych oraz za pomocą inteligencji roju. Na potrzeby rozwiązania tego problemu napisałem program w języku Python. Następnie porównałem wydajność oraz jakość napisanych przeze mnie sposobów na rozwiązanie tej zagadki.

## 2 Przedstawienie problemu

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. Poniżej znajduje się przykładowy rozwiązany nonogram.



# 3 Implementacja problemu

W celu rozwiązania zagadki Nonogram stworzyłem klasę Nonogram, przyjmującą w konstruktorze reguły z rzędów i kolumn. Reguły muszą być zapisane w formie tablicy z listami wypełnionymi cyframi oznaczającymi ilość zamalowanych kratek w rzędzie lub kolumnie.

```
class Nonogram:
    def __init__(self, rows, cols):
        self.rows = rows
        self.cols = cols
```

Klasa posiada dwie funkcje runGA oraz runSwarm. Funkcja runGA służy do uruchamiania algorytmu genetycznego i przyjmuje następujące argumenty:
- fitness_func - argument do wyboru funkcji oceniającej
- generations - ilość iteracji (pokoleń) algorytmu
- sol_per_pop - ilość chromosomów w populacji
- mutation_percent_genes - procent genów, które podlegają mutacji
- num_parents_mating - ilość rodziców wyłonionych do "rozmnażania"
fontsize

```
def runGA(self, fitness_func, generations, sol_per_pop, mutation_percent_genes,
    num_parents_mating):
```

Natomiast funkcja runSwarm służy do uruchaminia algorytmu inteligencji roju. Funkcja przyjmuje następujący argument:
- fitness_func - argument do wyboru funkcji oceniającej
- generations - ilość iteracji algorytmu
- c1 - cognitive parameter
- c2 - social parameter
- w - inertia
- k - liczba sąsiadów
- p - wartość 1 oznacza sumę wartości bezwzględnych, wartość 2 oznacza odległość euklidesową.

```
def runSwarm(self, fitness_func, generations, c1, c2, w, k=None, p=None):
```

# 4 Rozwiązanie problemu

## 4.1 Funkcje fitness

Na potrzeby rozwiązania problemu napisałem w języku programowania Python cztery funkcje fitness, które sumują ujemne punkty w zależności od tego jak dobre jest dane rozwiązanie.
Większość funkcji fitness korzysta z napisanej przeze mnie funkcji arrayEquals, która przyjmuje dwie tablice, a następnie sprawdza czy wszystkie elementy na tych samych indeksach są takie same.

```
def arrayEquals(tab1, tab2):
    if len(tab1) != len(tab2):
        return False
    for i in range(len(tab1)):
        if tab1[i] != tab2[i]:
            return False
    return True
```

Do uruchamiania algorytmów i robienia pomiarów napisałem funkcję, która uruchamia wybrane algorytm z podanymi parametrami 100 razy, a następnie wylicza i wyświetla rezultaty.

```
def measureTimeAndResult(func, args, counter=False):
    tabTimes = []
    tabResults = []
    tabBestSol = []
    solved = 0
    if counter:
```

```
        for i in range(100):
            start = time.time()
            temp = func(*args)
            tabTimes.append(time.time() - start)
            tabResults.append(temp['best_solution_generation'])
            tabBestSol.append(temp['solution_fitness'])
            if temp['solution_fitness'] == 0:
                solved += 1
            print(i, "/ 100")
    else:
        for i in range(100):
            start = time.time()
            temp = func(*args)
            tabTimes.append(time.time() - start)
            tabResults.append(temp['best_solution_generation'])
            tabBestSol.append(temp['solution_fitness'])
            if temp['solution_fitness'] == 0:
                solved += 1
    print("Średnia generacja:", mean(tabResults))
    print("Średni czas:", mean(tabTimes))
    print("Średnie najlepsze rozwiązanie:", mean(tabBestSol))
    print("Procent poprawnych rozwiązań:", solved)
    print()
```

Dodatkowo napisałem funkcję, która wylicza średnią z podanej tablicy z liczbami

```
def mean(tab):
    return sum(tab) / len(tab)
```

### 4.1.1 Podstawowa wersja funkcji fitness

Najbardziej podstawowa wersja funkcji fitness przyjmuje jako rozwiązanie ciąg liczb o długości ilości pól na planszy nonogramy (liczby mają wartości 0 dla niezamalowanego pola, 1 dla zamalowanego pola). Następnie ciąg liczb jest zamieniany w listę list, które reprezentują rzędy w nonogramie. Potem dla każdego rzędu i każdej kolumny są zliczane zamalowane pola w zapisywane w formie listy list. Przykładowo dla rzędu [0,1,1,0,1] zostanie utworzona lista [2,1]. Na koniec zostaje sprawdzone czy otrzymane listy są takie same jak listy podane w odpowiadających im regułach. Jeżeli listy się różnią to wtedy rozwiązanie otrzymuje ujemne punkty. Na koniec zwracana jest ilość ujemnych punktów. fontsize

```
def fitness(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:
        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)
    for i in range(len(solution[0])):
        counter = 0
        temp = []
        for j in range(len(solution[i])):
            if solution[j][i] == 1:
                counter += 1
            elif counter != 0:
```

```
                temp.append(counter)
                counter = 0
        if counter != 0:
            temp.append(counter)
        resultCol.append(temp)
    wrongCounter = 0
    for i in range(len(self.rows)):
        if not arrayEquals(self.rows[i], resultRow[i]):
            wrongCounter -= 1
        if not arrayEquals(self.cols[i], resultCol[i]):
            wrongCounter -= 1
    return wrongCounter
```

### 4.1.2 Ulepszona wersja funkcji fitness

Ulepszona wersja funkcji fitness początkowo działa bardzo podobnie do podstawowej wersji funkcji fitness. Funkcja przyjmuje jako rozwiązanie ciąg liczb o długości ilości pól na planszy i zamienia go w listę list, które reprezentują rzędy i kolumny w nonogramie. Potem dla każdego rzędu i każdej kolumny są zliczane zamalowane pola i zostają zapisane w formie listy list, tak samo jak w podstawowej wersji funkcji fitness. Następnie funkcja sprawdza czy długość listy odpowiadającej danemu rzędowi lub kolumnie jest równa długości listy odpowiadającej danej regule. W przeciwnym wypadku dodawane są ujemne punkty. Następnie funkcja sprawdza poszczególne elementy w liście odpowiadającej rzędowi lub kolumnie. Jeżeli wartości nie są równe z wartością w regule to wtedy dodawane są ujemne punkty za każdy nierówny element. W przypadku gdy listy nie są równej długości to pętla iteruje tylko do ostatniego elementu krótszej listy. Na koniec zwracana jest ilość ujemnych punktów.

```
def fitness_tuning(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:
        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)
    for i in range(len(solution[0])):
        counter = 0
        temp = []
        for j in range(len(solution[i])):
            if solution[j][i] == 1:
                counter += 1
            elif counter != 0:
                temp.append(counter)
                counter = 0
        if counter != 0:
            temp.append(counter)
        resultCol.append(temp)
    wrongCounter = 0
    for i in range(len(self.rows)):
        if len(self.rows[i]) == len(resultRow[i]):
            for j in range(len(self.rows[i])):
                if self.rows[i][j] != resultRow[i][j]:
                    wrongCounter -= 1
        else:
            wrongCounter -= math.fabs(len(self.rows[i]) - len(resultRow[i]))
            if len(self.rows[i]) > len(resultRow[i]):
                for j in range(len(resultRow[i])):
                    if self.rows[i][j] != resultRow[i][j]:
                        wrongCounter -= 1
```

```
            else:
                for j in range(len(self.rows[i])):
                    if self.rows[i][j] != resultRow[i][j]:
                        wrongCounter -= 1
        if len(self.cols[i]) == len(resultCol[i]):
            for j in range(len(self.cols[i])):
                if self.cols[i][j] != resultCol[i][j]:
                    wrongCounter -= 1
        else:
            wrongCounter -= math.fabs(len(self.cols[i]) - len(resultCol[i]))
            if len(self.cols[i]) > len(resultCol[i]):
                for j in range(len(resultCol[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
            else:
                for j in range(len(self.cols[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
    return wrongCounter
```

### 4.1.3 Funkcja fitness rozmieszczająca gotowe bloki reguł na planszy

Funkcja fitness przyjmuje jako rozwiązanie ciąg liczb o długości ilości wszystkich elementów w regułach dla rzędów i kolumn. Liczby mają wartości od 0 do wartości długości każdego rzędu i kolumny (dla nonogramu 5x5 będą to wartości od 0 do 4). Do zmiennej tab jest przypisywana lista list z liczbami 0 reprezentująca rzędy w nonogramie. Następnie funkcja iteruje po rzędach, a potem po kolumnach malując odpowiednie pola zgodnie z pozycją w rozwiązaniu i ilością pól podaną w regule. Przykładowo dla nonogramu 3x3 jeżeli reguła będzie [1,1], a w rozwiązaniu dla danego rzędu lub kolumny będą liczby [0,2] to zostanie zamalowany pierwszy i ostatni element w danym rzędzie lub kolumnie. Inny przykład z nonogramem 5x5. Jeżeli reguła będzie [2,1], a rozwiązaniem dla danego rzędu lub kolumny będą wartości [1,4] to zostanie zamalowane 2, 3 oraz 5 pole. Dalej funkcja wygląda podobnie jak w przypadku ulepszonej wersji funkcji fitness. Wartości zmiennej tab są zamieniane w listę list, które reprezentują zamalowane pola w rzędach i kolumnach w nonogramie. Funkcja dodaje ujemny punkty za różne długości list reguł i list reprezentujących rzędy i kolumny. Potem są dodawane ujemne punkty za każdy różniący się element w regule i w liście reprezentującą rzędy i kolumny. Na koniec zwracana jest ilość ujemnych punktów.

```
def fitness_another(solution, solution_idx):
    tab = [[0 for i in range(len(self.cols))] for j in range(len(self.rows))]
    iterator = 0
    wrongCounter = 0
    solution = solution.astype(int)
    for i in range(len(self.rows)):
        for j in range(len(self.rows[i])):
            for k in range(self.rows[i][j]):
                if solution[iterator] + k < len(self.rows):
                    tab[i][solution[iterator] + k] = 1
                else:
                    wrongCounter -= 1
            iterator += 1
    for i in range(len(self.cols)):
        for j in range(len(self.cols[i])):
            for k in range(self.cols[i][j]):
                if solution[iterator] + k < len(self.cols):
                    tab[solution[iterator] + k][i] = 1
                else:
                    wrongCounter -= 1
            iterator += 1
    resultRow = []
    resultCol = []
    solution = tab
    for i in solution:
        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)
    for i in range(len(solution[0])):
        counter = 0
        temp = []
        for j in range(len(solution[i])):
            if solution[j][i] == 1:
                counter += 1
            elif counter != 0:
                temp.append(counter)
                counter = 0
        if counter != 0:
            temp.append(counter)
        resultCol.append(temp)
    for i in range(len(self.rows)):
        if len(self.rows[i]) == len(resultRow[i]):
            for j in range(len(self.rows[i])):
                if self.rows[i][j] != resultRow[i][j]:
                    wrongCounter -= 1
        else:
            wrongCounter -= math.fabs(len(self.rows[i]) - len(resultRow[i]))
            if len(self.rows[i]) > len(resultRow[i]):
                for j in range(len(resultRow[i])):
                    if self.rows[i][j] != resultRow[i][j]:
                        wrongCounter -= 1
            else:
                for j in range(len(self.rows[i])):
                    if self.rows[i][j] != resultRow[i][j]:
                        wrongCounter -= 1
        if len(self.cols[i]) == len(resultCol[i]):
            for j in range(len(self.cols[i])):
                if self.cols[i][j] != resultCol[i][j]:
                    wrongCounter -= 1
        else:
            wrongCounter -= math.fabs(len(self.cols[i]) - len(resultCol[i]))
            if len(self.cols[i]) > len(resultCol[i]):
                for j in range(len(resultCol[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
            else:
                for j in range(len(self.cols[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
    return wrongCounter
```

### 4.1.4 Funkcja fitness rozmieszczająca gotowe bloki reguł na planszy tylko w rzędach

Funkcja fitness przyjmuje jako rozwiązanie ciąg liczb o długości ilości wszystkich elementów w regułach dla rzędów. Liczby mają wartości od 0 do wartości długości każdego rzędu (dla nonogramu 5x5 będą to wartości od 0 do 4). Do zmiennej tab jest przypisywana lista list z liczbami 0 reprezentująca rzędy w nonogramie. Następnie funkcja iteruje po rzędach malując odpowiednie pola zgodnie z pozycją podaną w rozwiązaniu i ilością pól podaną w regule. Wartości zmiennej tab są zamieniane w listę list, które reprezentują zamalowane pola w rzędach i kolumnach w nonogramie. Funkcja dodaje ujemny punkty za różne długości list reguł i list reprezentujących kolumny. Potem są dodawane ujemne punkty za każdy różniący się element w regule i w liście reprezentującej kolumny. Na koniec zwracana jest ilość ujemnych punktów.

```
def fitness_another_tuning(solution, solution_idx):
    tab = [[0 for i in range(len(self.cols))] for j in range(len(self.rows))]
    iterator = 0
    wrongCounter = 0
    solution = solution.astype(int)
    for i in range(len(self.rows)):
        for j in range(len(self.rows[i])):
            for k in range(self.rows[i][j]):
```

```
                if solution[iterator] + k < len(self.rows):
                    tab[i][solution[iterator] + k] = 1
                else:
                    wrongCounter -= 1
            iterator += 1
    resultCol = []
    solution = tab
    for i in range(len(solution[0])):
        counter = 0
        temp = []
        for j in range(len(solution[i])):
            if solution[j][i] == 1:
                counter += 1
            elif counter != 0:
                temp.append(counter)
                counter = 0
        if counter != 0:
            temp.append(counter)
        resultCol.append(temp)
    for i in range(len(self.cols)):
        if len(self.cols[i]) == len(resultCol[i]):
            for j in range(len(self.cols[i])):
                if self.cols[i][j] != resultCol[i][j]:
                    wrongCounter -= 1
        else:
            wrongCounter -= math.fabs(len(self.cols[i]) - len(resultCol[i]))
            if len(self.cols[i]) > len(resultCol[i]):
                for j in range(len(resultCol[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
            else:
                for j in range(len(self.cols[i])):
                    if self.cols[i][j] != resultCol[i][j]:
                        wrongCounter -= 1
    return wrongCounter
```

## 4.2 Algorytm genetyczny

Dla algorytmu genetycznego ustawiłem następujące parametry:

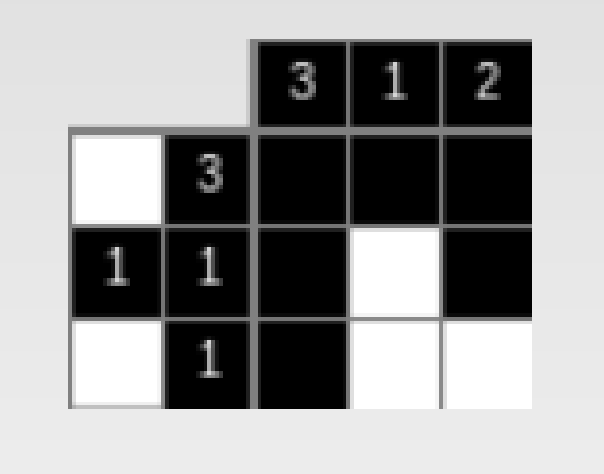
```
generations = 100
sol_per_pop = 100
num_parents_mating = 50
keep_parents = 2
parent_selection_type = "sss"
crossover_type = "single_point"
mutation_type = "random"
```

Ilość pokoleń w algorytmie ustawiłem na 100. Ilość chromosomów w populacji ustawiłem na 100. Procent rodziców, które zostaje zachowane ustawiłem na 2.Ilość rodziców, których wybieram do populacji ustawiłem na 50. Sposób wyboru rodziców ustawiłem na "sss" czyli "steady-state selection". Rodzaj krzyżowania ustawiłem na single_point. Typ mutacji ustawiłem na losowy.
Dla każdego rozmiaru nonogramu i rodzaju funkcji fitness ustawiałem inny procent szansy na mutację. Dla każdego rozmiaru nonogramu i każdej funkcji fitness uruchomiłem po 100 prób, a następnie z otrzymanych wyników wyliczyłem średnie.
Skróty dla nazw funkcji:
- PF - podstawowa wersja funkcji fitness
- UF - ulepszona wersja funkcji fitness
- PB - funkcja fitness rozmieszczająca gotowe bloki reguł na planszy
- UB - funkcja fitness rozmieszczająca gotowe bloki reguł na planszy tylko w rzędach

Poniżej znajdują się testy dla różnych rozmiarów zagadki nonogram. Informacje o wynikach są umieszczone w tabelach w których umieściłem informacje na temat optymalnej wybranej przeze mnie szansy na mutacje, średniej wartości pokolenia, po którym algorytm genetyczny osiągnął najlepsze rozwiązanie, średni czas działania algorytmu genetycznego, średnia wartość zwracana przez funkcje fitness dla najlepszego rozwiązania (im wartość jest bliższa 0 tym lepsze dopasowanie rozwiązania) oraz procent poprawnych rozwiązań (gdy wartość fitness najlepszego rozwiązania była równa 0), na 100 uruchomień algorytmu. Czasy działania algorytmów są podanie w sekundach.

### 4.2.1 Nonogram 3x3

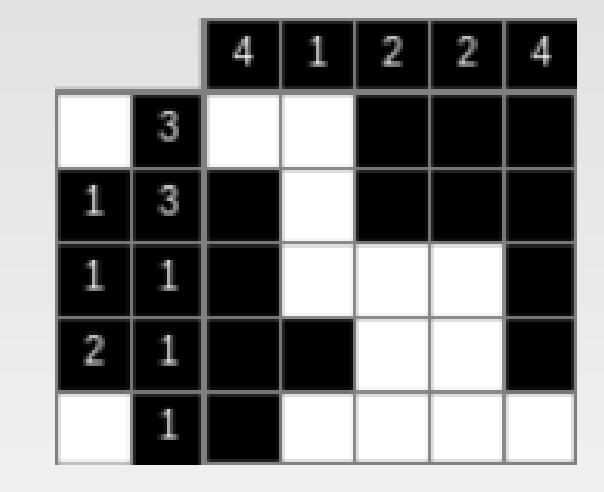
Wybrałem nonogram o rozmiarze 3x3, który wygląda tak:




| Nazwa funkcji | Szansa na mutacje | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|---|
| PF | 12 | 1.59 | 0.389 | 0 | 100% |
| UF | 12 | 1.88 | 0.407 | 0 | 100% |
| PB | 15 | 1.65 | 0.462 | 0 | 100% |
| UB | 34 | 0.01 | 0.382 | 0 | 100% |

### 4.2.2 Nonogram 5x5

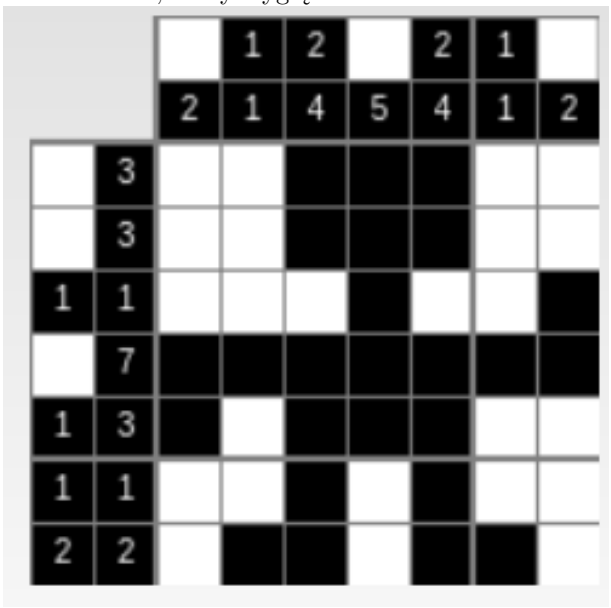
Wybrałem nonogram o rozmiarze 5x5, który wygląda tak:




| Nazwa funkcji | Szansa na mutacje | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|---|
| PF | 5 | 24.3 | 0.531 | -0.1 | 90% |
| UF | 5 | 27.23 | 0.563 | 0 | 100% |
| PB | 8 | 21.04 | 0.663 | -0.08 | 96% |
| UB | 13 | 15.14 | 0.502 | 0 | 100% |

### 4.2.3 Nonogram 7x7

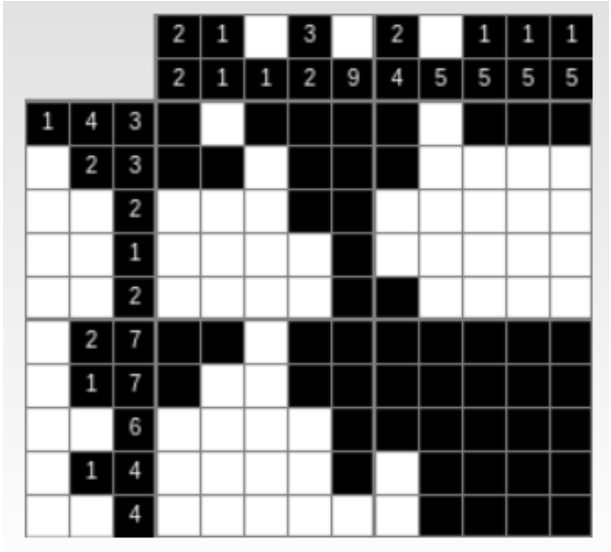
Wybrałem nonogram o rozmiarze 7x7, który wygląda tak:




| Nazwa funkcji | Szansa na mutacje | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|---|
| PF | 5 | 60.87 | 0.79 | -3.43 | 1% |
| UF | 5 | 64.95 | 0.828 | -4.01 | 3% |
| PB | 5 | 37.98 | 0.972 | -0.38 | 82% |
| UB | 7 | 25.73 | 0.694 | 0 | 100% |

### 4.2.4 Nonogram 10x10

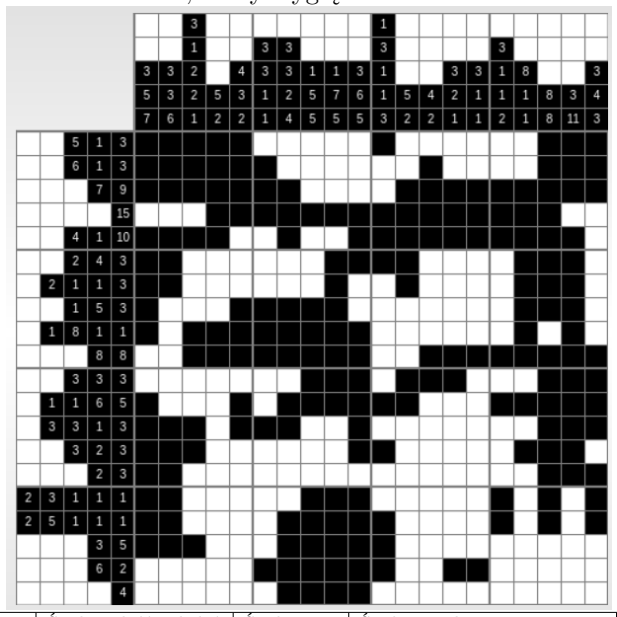
Wybrałem nonogram o rozmiarze 10x10, który wygląda tak:




| Nazwa funkcji | Szansa na mutacje | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|---|
| PF | 5 | 59.53 | 1.093 | -12.88 | 0% |
| UF | 5 | 66.6 | 1.184 | -21.43 | 0% |
| PB | 5 | 77.22 | 1.348 | -7.39 | 2% |
| UB | 7 | 60.41 | 0.874 | -2.54 | 7% |

### 4.2.5 Nonogram 20x20

Wybrałem nonogram o rozmiarze 20x20, który wygląda tak:



| Nazwa funkcji | Szansa na mutacje | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|---|
| PF | 5 | 41.75 | 3.38 | -37.36 | 0% |
| UF | 5 | 70.93 | 3.612 | -129.49 | 0% |
| PB | 5 | 84.43 | 4.257 | -87.88 | 0% |
| UB | 5 | 85.83 | 2.349 | -29.83 | 0% |

## 4.3 Optymalizacja rojem cząsteczek

Skróty dla nazw funkcji:
- PF - podstawowa wersja funkcji fitness
- UF - ulepszona wersja funkcji fitness
- PB - funkcja fitness rozmieszczająca gotowe bloki reguł na planszy
- UB - funkcja fitness rozmieszczająca gotowe bloki reguł na planszy tylko w rzędach

Dla funkcji PF i UF wykorzystałem algorytm binarnej optymalizacji roju cząstek (binarny PSO). Algorytm pobiera zestaw rozwiązań kandydujących i próbuje znaleźć najlepsze rozwiązanie za pomocą metody aktualizacji pozycji i prędkości. Przekształciłem ten algorytm w taki sposób aby zwracał wartości 0 lub 1.
Natomiast dla funkcji PB i UB skorzystałem z globalnego najlepszego algorytmu optymalizacji roju cząstek (gbest PSO). Algorytm pobiera zestaw rozwiązań kandydujących i próbuje znaleźć najlepsze rozwiązanie za pomocą metody aktualizacji pozycji i prędkości. Używa topologii gwiazdy, w której każda cząsteczka jest przyciągana do cząsteczki o najlepszych wynikach. Przekształciłem ten algorytm w taki sposób aby zwracał liczby całkowite od 0 do wartości potrzebnej dla danej funkcji (ilość wartości w regułach dla rzędów lub dla rzędów i kolumn).
Dla optymalizacji rojem cząsteczek dla funkcji fitness PF i UF ustawiłem następujące parametry:

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

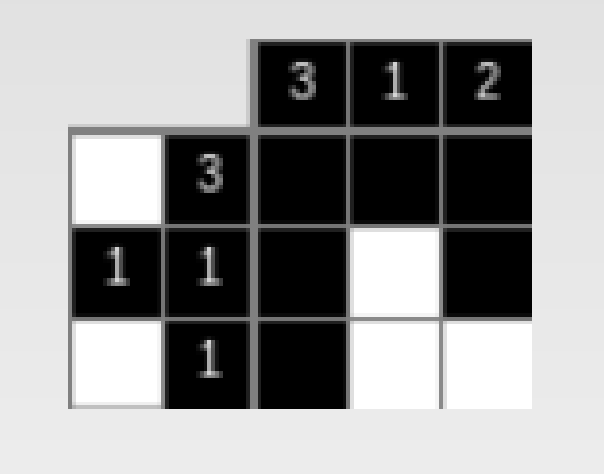
Natomiast dla optymalizacji rojem cząsteczek dla funkcji fitness PB i UB ustawiłem następujące parametry:

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

Wyjaśnienia znaczenia parametrów algorytmu:
dimensions - ilość wymiarów w przestrzeni cząstek
iters - ilość iteracji algorytmu
c1 - (cognitive parameter) określa jak bardzo jednostka będzie dążyła do najlepszego indywidualnego rozwiązania.
c2 - (social parameter) określa jak mocno jednostka będzie dążyła do najlepszego lokalnego rozwiązania.
w - (inertia) jest to wartość wpływu poprzedniej prędkości.
k - jest liczbą sąsiadów, z którymi "komunikuje się" pojedyncza jednostka.
p - wartość 1 oznacza sumę wartości bezwzględnych, wartość 2 oznacza odległość euklidesową.
Poniżej znajdują się testy dla różnych rozmiarów zagadki nonogram. Informacje o wynikach są umieszczone w tabelach w których umieściłem informacje na temat średniej wartości pokolenia, po którym algorytm genetyczny osiągnął najlepsze rozwiązanie, średni czas działania algorytmu genetycznego, średnia wartość zwracana przez funkcje fitness dla najlepszego rozwiązania (im wartość jest bliższa 0 tym lepsze dopasowanie rozwiązania) oraz procent poprawnych rozwiązań (gdy wartość fitness najlepszego rozwiązania była równa 0), na 100 uruchomień algorytmu. Czasy działania algorytmów są podanie w sekundach.

### 4.3.1 Nonogram 3x3

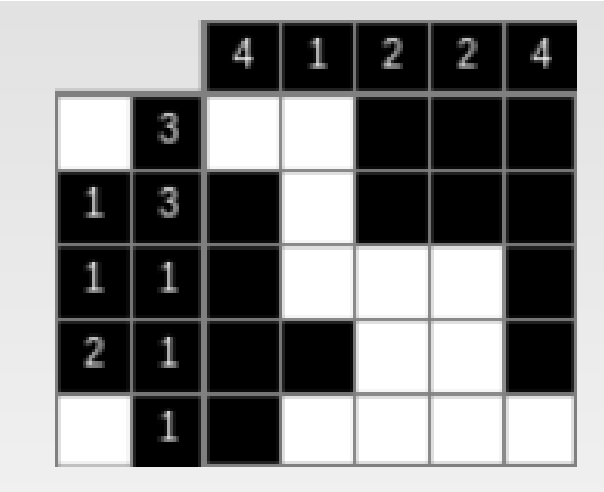
Wybrałem nonogram o rozmiarze 3x3, który wygląda tak:




| Nazwa funkcji | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|
| PF | 189.95 | 0.154 | 1.18 | 45% |
| UF | 157.01 | 0.152 | 1 | 51% |
| PB | 130.86 | 0.119 | 1.3 | 59% |
| UB | 49.26 | 0.122 | 0.85 | 72% |

### 4.3.2 Nonogram 5x5

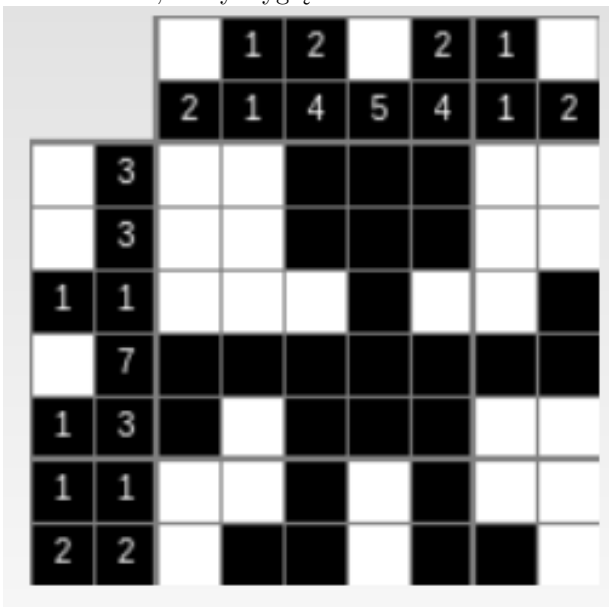
Wybrałem nonogram o rozmiarze 5x5, który wygląda tak:




| Nazwa funkcji | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|
| PF | 373.27 | 0.185 | 4.57 | 0% |
| UF | 442.23 | 0.202 | 4.97 | 2% |
| PB | 234.31 | 0.148 | 7.88 | 0% |
| UB | 152.33 | 0.121 | 3.33 | 4% |

### 4.3.3 Nonogram 7x7

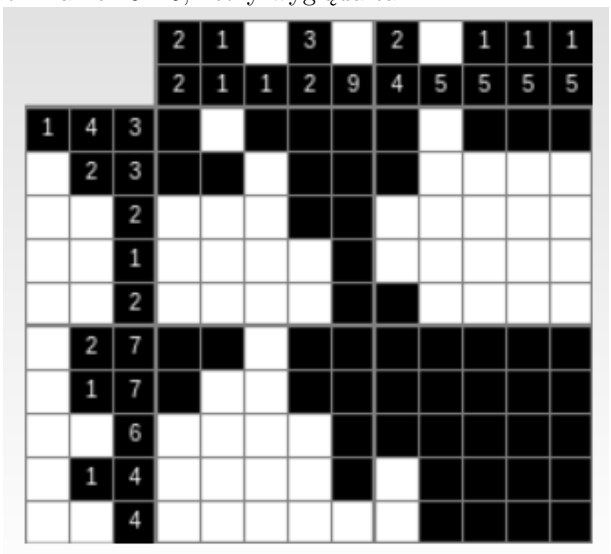
Wybrałem nonogram o rozmiarze 7x7, który wygląda tak:




| Nazwa funkcji | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|
| PF | 480.1 | 0.216 | 8.52 | 0% |
| UF | 569.45 | 0.215 | 10.83 | 0% |
| PB | 287.92 | 0.167 | 15.59 | 0% |
| UB | 266.52 | 0.144 | 8.61 | 0% |

### 4.3.4 Nonogram 10x10

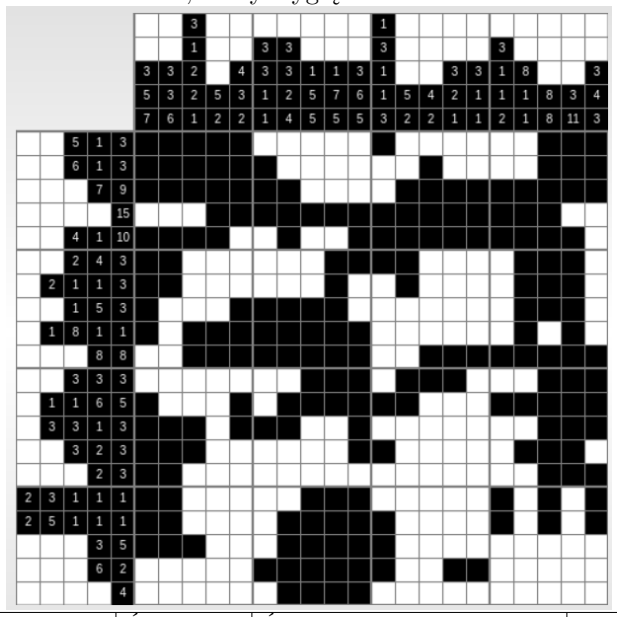
Wybrałem nonogram o rozmiarze 10x10, który wygląda tak:




| Nazwa funkcji | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|
| PF | 322.67 | 0.28 | 16.14 | 0% |
| UF | 637.38 | 0.278 | 21.77 | 0% |
| PB | 333.52 | 0.214 | 31.51 | 0% |
| UB | 320.63 | 0.166 | 15.44 | 0% |

### 4.3.5 Nonogram 20x20

Wybrałem nonogram o rozmiarze 20x20, który wygląda tak:



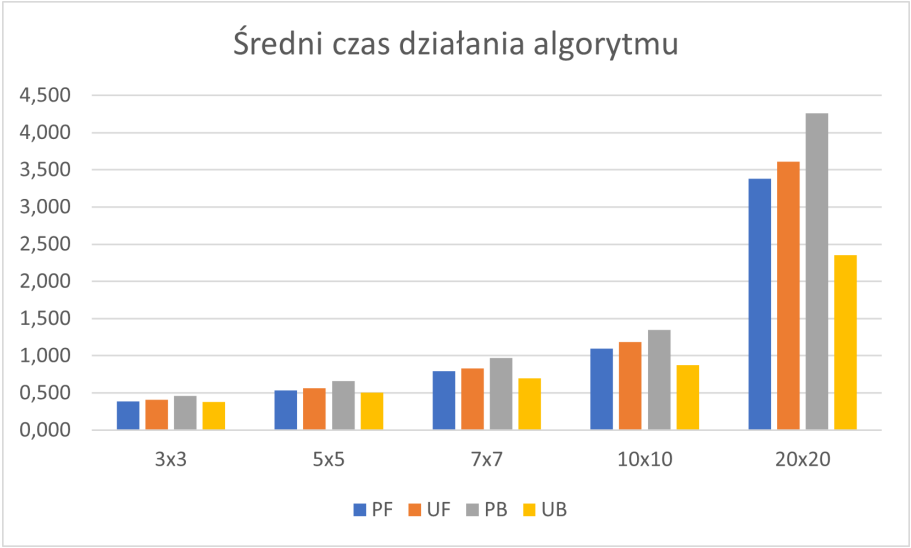
| Nazwa funkcji | Średnia ilość pokoleń | Średni czas | Średnie najlepsze rozwiązanie | Procent poprawnych rozwiązań |
|---|---|---|---|---|
| PF | 275.21 | 0.802 | 39.03 | 0% |
| UF | 852.35 | 0.811 | 95.61 | 0% |
| PB | 365.31 | 0.447 | 121.73 | 0% |
| UB | 354.78 | 0.281 | 64.96 | 0% |

# 5 Porównanie wyników

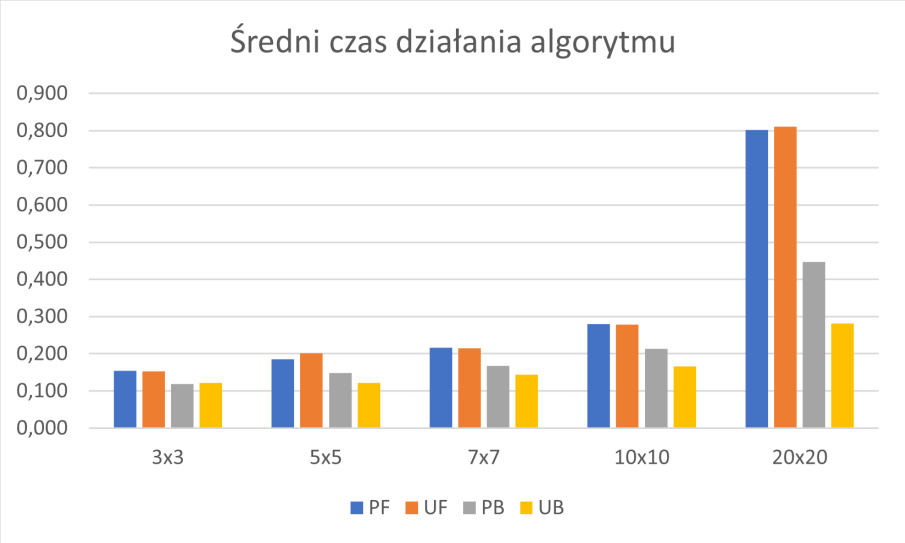
## 5.1 Porównanie czasów działania algorytmu

Czasy działania algorytmów są podanie w sekundach.

Algorytm genetyczny



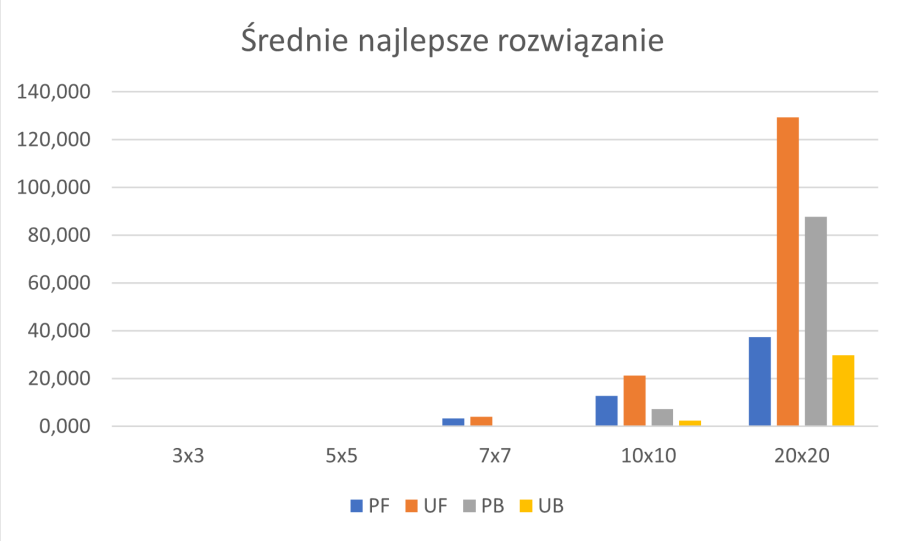

Optymalizacja rojem cząstek



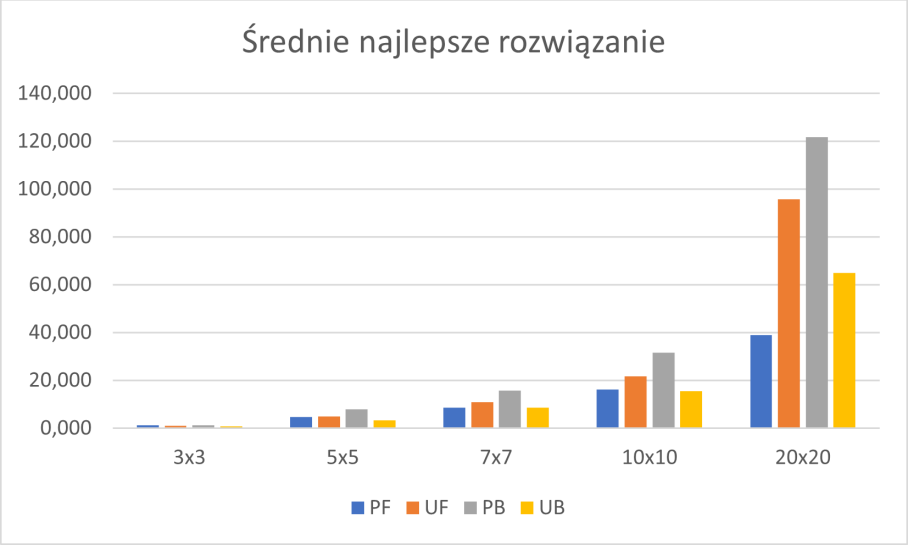

## 5.2 Porównanie wartości zwracanych przez funkcję fitness

Funkcja fitness ocenia czy podanie rozwiązanie jest bliższe lub dalsze od poprawnego rozwiązania. Im większa wartość na wykresie, tym rozwiązanie jest mniej poprawne.

Algorytm genetyczny

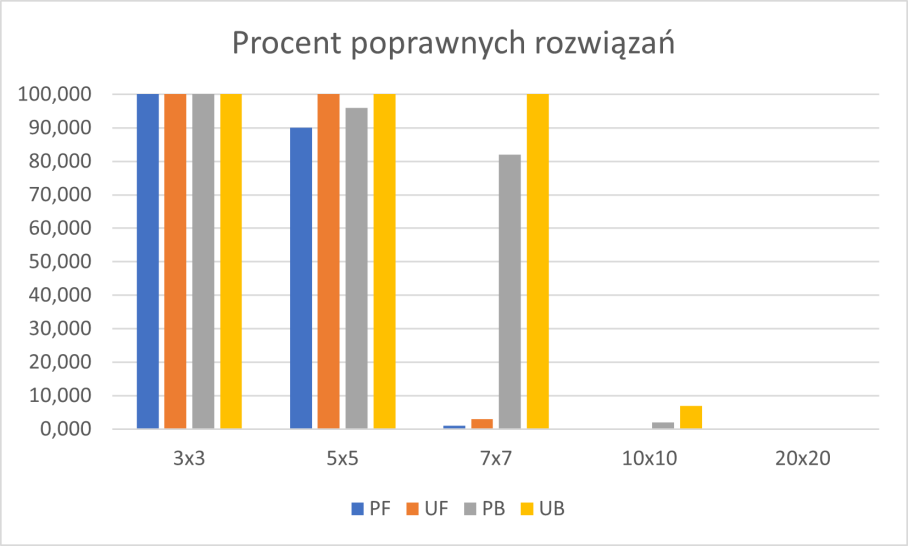



Optymalizacja rojem cząstek

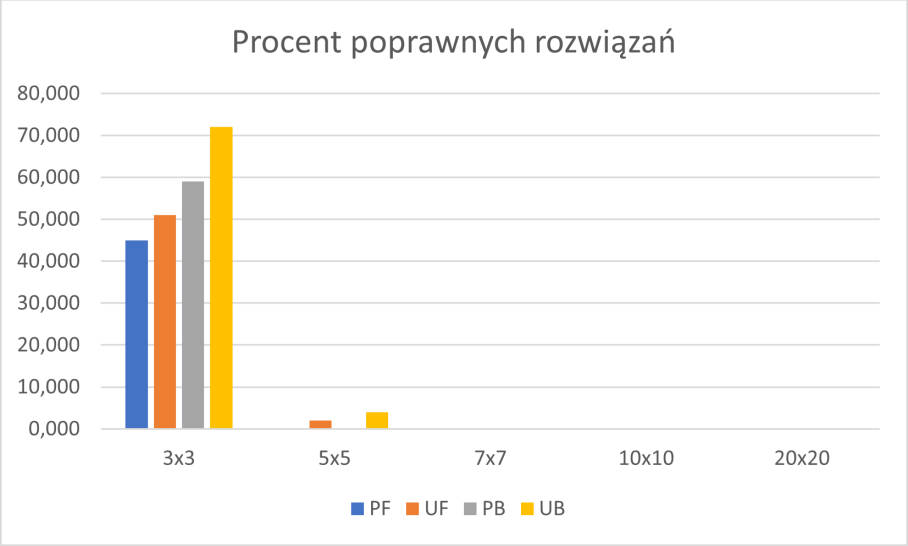



## 5.3 Porównanie ilości poprawnych rozwiązań na 100 iteracji

Algorytm genetyczny




Optymalizacja rojem cząstek



# 6 Podsumowanie

## 6.1 Algorytmy genetyczne

Na podstawie powyższych wyników obliczeń można stwierdzić, że algorytmy genetyczne niezbyt dobrze sobie radzą ze znajdowaniem rozwiązań dla zagadek typu nonogram. Podstawowa funkcja fitness oraz ulepszona wersja podstawowej funkcji fitness działały zadowalająco jedynie dla nonogramów do wielkości 5x5. Natomiast funkcje fitness, które rozmieszczały gotowe bloki na planszy działały zadowalająco dla nonogramów o maksymalnej wielkości 7x7. Średni czas działania algorytmu rósł wraz z wzrostem wielkości nonogramu. Funkcja fitness rozmieszczająca gotowe bloki na planszy działa trochę wolniej od pozostałych funkcji. Najszybciej działa funkcja fitness rozmieszczająca gotowe bloki tylko w rzędach.

## 6.2 Optymalizacja rojem cząsteczek

Na podstawie powyższych wyników obliczeń można stwierdzić, że optymalizacja rojem cząstek bardzo słabo radzi sobie ze znajdowaniem rozwiązań dla zagadek typu nonogram. Optymalizacja rojem cząsteczek znajdowała całkiem zadowalające rozwiązania jedynie dla nonogramów o rozmiarze 3x3, przy większych rozmiarach trudno było znaleźć idealne rozwiązanie. Średni czas działania rósł wraz z wzrostem wielkości nonogramu. Podstawowa wersja funkcji fitness oraz ulepszona wersja podstawowej wersji funkcji fitness działały najwolniej. Natomiast najszybciej działa funkcja fitness rozmieszczająca gotowe bloki tylko w rzędach.

## 6.3 Porównanie algorytmów

Algorytm genetyczny okazał się dokładniejszy i bardziej skuteczny. Znajdował poprawne rozwiązania dla większych rozmiarów nonogramów. Natomiast optymalizacja rojem cząstek pomimo mniejszej skuteczności, szybciej znajdowała rozwiązania (nawet ze zwiększoną 10-krotnie ilością iteracji).

# 7 Bibliografia


- https://pygad.readthedocs.io/en/latest/
- https://pyswarms.readthedocs.io/en/latest/
- https://nonograms.relaxpuzzles.com
- https://pl.wikipedia.org/wiki/Obrazek_logiczny