Rozwiązywanie problemu Kuromasu za pomocą algorytmów genetycznych i rojów cząstek.

Igor Pustovoy

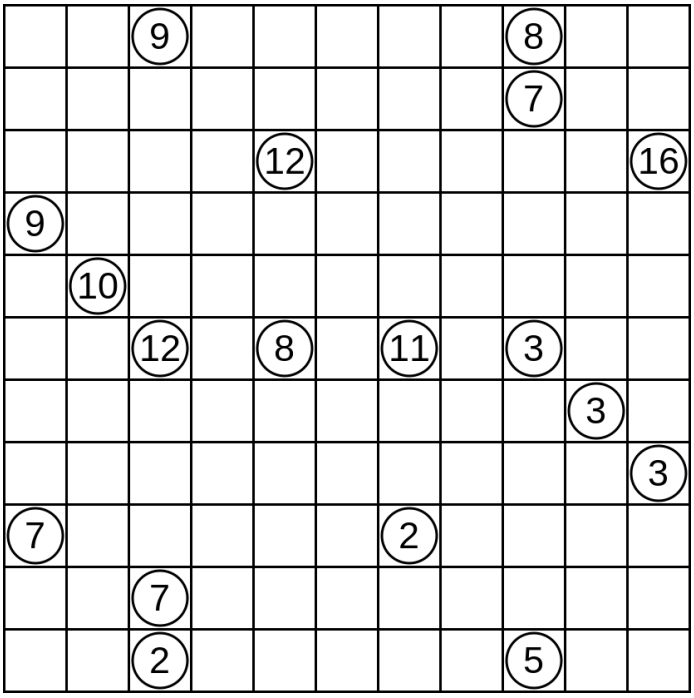
Kwiecień 2022

1 Wprowadzenie

Kuromasu, czyli "Gdzie są czarne pola" to japońska gra wydana przez Nikoli, po raz pierwszy ujrzała światło dziennie w 1991 roku.

1.1 Zasady

Gracz rozpoczyna grę widząc prostokątną planszę, na której w niektórych miejscach umieszczone są liczby. Przykład takiej planszy widnieje poniżej.

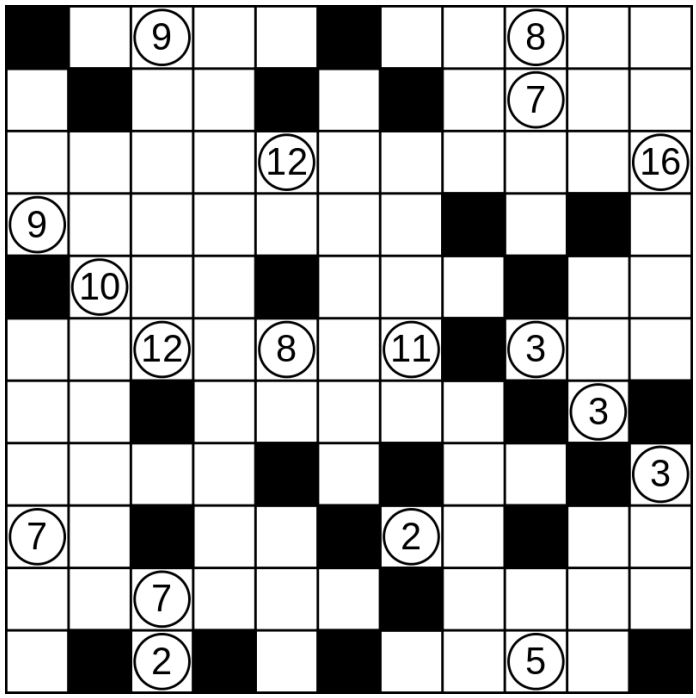


Następnie, jego zadaniem jest zamalowanie na czarno wybranych pól, tak aby każdemu polu z liczbą odpowiadała ilość białych pól w poziomie oraz pionie od tego pola, równa tej liczbie. Jeżeli jednak w trakcie liczenia napotkamy na koniec planszy albo na czarne pole, to nie możemy liczyć dalej.

Wyodrębniamy jeszcze 3 podstawowe zasady:

- Pola z liczbami muszą być białe.
- Czarne pola nie mogą się bezpośrednio dotykać.
- Każde białe pole musi łączyć się z przynajmniej jednym innym. Pionowo, poziomo lub na skos.

Rozwiązana zagadka znajduje się poniżej



2 Funkcje Fitness

Chromosomem będzie tablica długości sumy pól na planszy, składająca się jedynie z zer i jedynek.

0 = białe pole, 1 = czarne pole

Chromosom na czas działania funkcji przekształcimy w tablicę dwuwymiarową korzystając z biblioteki numpy.

```
# Reshaping a 1D chromosome array into a 2D one.
solution2d = np.reshape(solution, (len(grid), len(grid[0])))
```

2.1 Podstawowa Funkcja Fitness

Funkcja fitness będzie zwracała ujemną ilość pól z liczbami, które mają poprawną wartość liczbową. Idealny wynik to więc 0. Jeżeli inne zasady będą złamane to w tej wersji funkcji zwracamy ujemną długość chromosomu, czyli ilość pól na planszy.

2.1.1 Implementacja

Zaczynamy od sprawdzenia czy żadne z czarnych pól nie leży bezpośrednio przy innym. W tej wersji funkcji Fitness każdemu chromosomowi łamiącemu tę zasadę dajemy najniższy możliwy wynik.

```
for i in range(0, len(solution2d)):
    for j in range(0, len(solution2d[0])):
        if solution2d[i][j] == 1:
            correct_placement = check_black_square_surroundings(i, j)
            if not correct_placement:
                return -len(solution)
```

```python
def check_black_square_surroundings(x, y):
    if x > 0 and solution2d[x - 1][y] == 1:
        return False
    if x < grid_height - 1 and solution2d[x + 1][y] == 1:
        return False
    if y > 0 and solution2d[x][y - 1] == 1:
        return False
    if y < grid_width - 1 and solution2d[x][y + 1] == 1:
        return False
    return True
```

Jeżeli wiemy już, że czarne pola nie przylegają ze sobą, to nie musimy sprawdzać czy każde białe pole ma przynajmniej jedno inne białe pole w swoim otoczeniu, ponieważ jest to równoznaczne.

Następnie możemy przejść do analizy, czy wartości numerycznych pól odpowiadają białym polom pionowo i poziomo od nich. Sprawdzamy również czy żadne pole z liczbą nie jest zamalowane na czarno. Jeżeli jest, to zwracamy najniższy możliwy wynik.

```python
# Check if numbered squares have correct values
numbers_wrong = 0

for i in range(0, len(grid)):
    for j in range(0, len(grid[0])):
        if grid[i][j] != 0:
            if solution2d[i][j] == 1:
                return -len(solution)  # if not white
            if not check_number(i, j):
                numbers_wrong += 1
```

```
def check_number(x, y):

    def check_left(x1, y1):
        white_fields = 0
        y1 -= 1  # We don't want to count the starting square
                                                    multiple times
        while 0 <= y1 < grid_width and solution2d[x1][y1] == 0:
            white_fields += 1
            y1 -= 1
        return white_fields

    def check_right(x1, y1):
        white_fields = 0
        y1 += 1
        while 0 <= y1 < grid_width and solution2d[x1][y1] == 0:
            white_fields += 1
            y1 += 1
        return white_fields

    def check_up(x1, y1):
        white_fields = 0
        x1 -= 1
        while 0 <= x1 < grid_height and solution2d[x1][y1] == 0:
            white_fields += 1
            x1 -= 1
        return white_fields

    def check_down(x1, y1):
        white_fields = 0
        x1 += 1
        while 0 <= x1 < grid_height and solution2d[x1][y1] == 0:
            white_fields += 1
            x1 += 1
        return white_fields
```

Dla każdego pola z wartością liczbową wywołujemy funkcję check_number(). W zależności od wyniku inkrementujemy lub nie zmienną numbers_wrong.

Na końcu funkcji Fitness zwracamy ujemną wartość zmiennej numbers_wrong.

2.2 Funkcja Fitness z punktami karnymi

W tej wersji funkcji Fitness nie będziemy od razu odrzucać chromosomów, które łamią jakąś zasadę (wyjątkiem jest przypadek kiedy pole z liczbą jest czarne), ale będziemy przyznawać ujemne punkty odpowiadające stopniowi złamania danej zasady. Metoda karania została zapożyczona z pracy magisterskiej "Comparing Methods for Solving Kuromasu Puzzles" Tima Von Meurs.

2.2.1 Implementacja

Za każde jedno pole różnicy między wartością liczbową, a ilością pól w zasięgu widzenia będzie naliczany jeden punkt karny.

```
directions = [check_left, check_right, check_up, check_down]
for direction in directions:
    white_fields_sum += direction(x, y)

return abs(number - white_fields_sum)
```

Wszystkie kary będą ze sobą sumowane

```
# Check if numbered squares have correct values
numbers_penalty = 0

for i in range(0, len(grid)):
    for j in range(0, len(grid[0])):
        if grid[i][j] != 0:
            if solution2d[i][j] == 1:
                return -len(solution)  # Numbered squares must be white
            numbers_penalty += check_number(i, j)
```

Każde czarne pole będzie mogło dostać do 4 punktów karnych w zależności od ilości czarnych pól, które do niego przylegają.

```python
def check_black_square_surroundings(x, y):
    penalty = 0
    if x > 0 and solution2d[x - 1][y] == 1:
        penalty += 1
    if x < grid_height - 1 and solution2d[x + 1][y] == 1:
        penalty += 1
    if y > 0 and solution2d[x][y - 1] == 1:
        penalty += 1
    if y < grid_width - 1 and solution2d[x][y + 1] == 1:
        penalty += 1
    return penalty
```

Wszystkie kary będą ze sobą sumowane

```python
# Check for wrongly placed black squares
black_squares_penalty = 0

for i in range(0, len(solution2d)):
    for j in range(0, len(solution2d[0])):
        if solution2d[i][j] == 1:
            black_squares_penalty += check_black_square_surroundings(i, j)
```

Na końcu funkcji zwracana będzie ujemna łączna suma kar

```python
return -(numbers_penalty + black_squares_penalty)
```

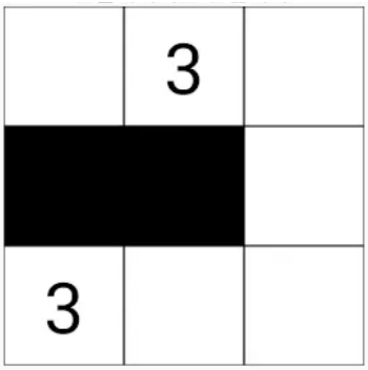
3 Algorytm genetyczny

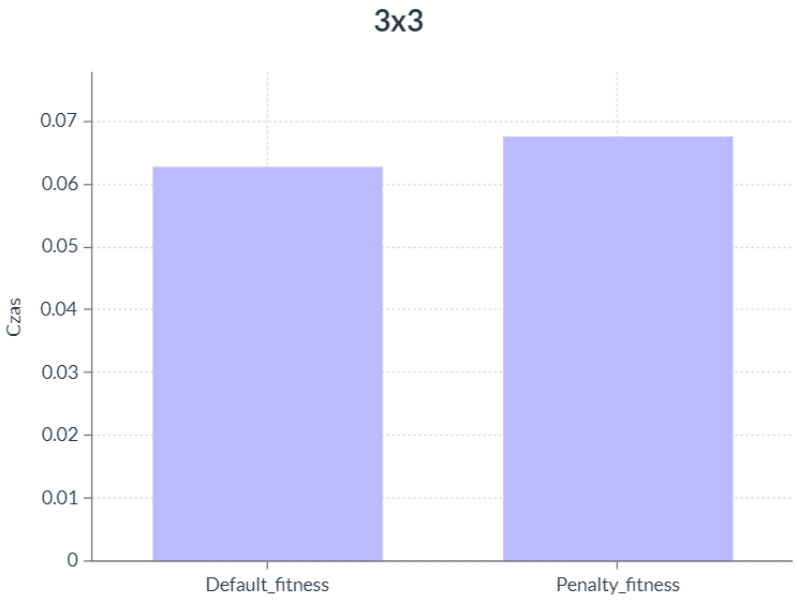
Po przetestowaniu różnych ustawień algorytmu zdecydowałem się na poniższe:

```
sol_per_pop = 600
num_genes = len(grid) * len(grid[0])

num_parents_mating = 300
num_generations = 800
keep_parents = 15

parent_selection_type = "sss"

crossover_type = "single_point"

mutation_type = "random"
mutation_percent_genes = 130 / (len(grid) * len(grid[0]))
```

Na różnej wielkości planszach będziemy testować obie wersje funkcji Fitness. Dla każdej wyliczona zostanie średnia arytmetyczna ze stu powtórzeń algorytmu.

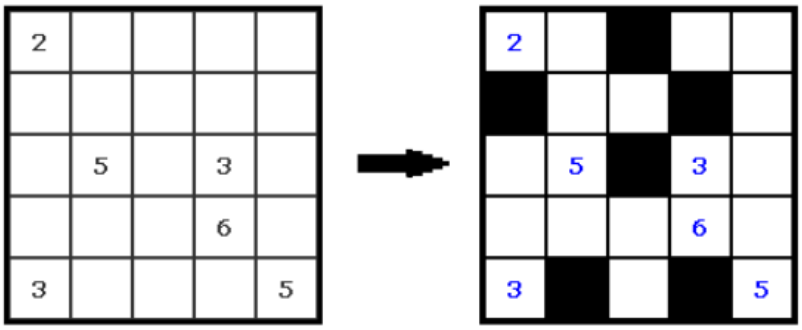
3.1 Plansza 3x3

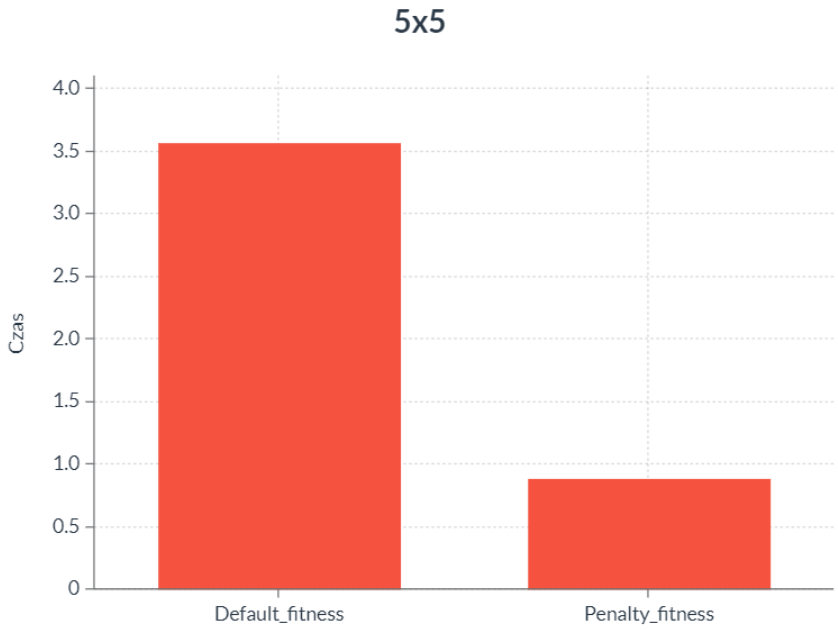




Na wykresie widzimy, że podstawowa funkcja Fitness średnio wykonała algorytm odrobinę szybciej niż jej wersja z karami. Dla podstawowej wersji średni czas wyniósł: 0.06286500692367554 sekund, natomiast dla rozszerzonej: 0.06770203828811645 sekund.

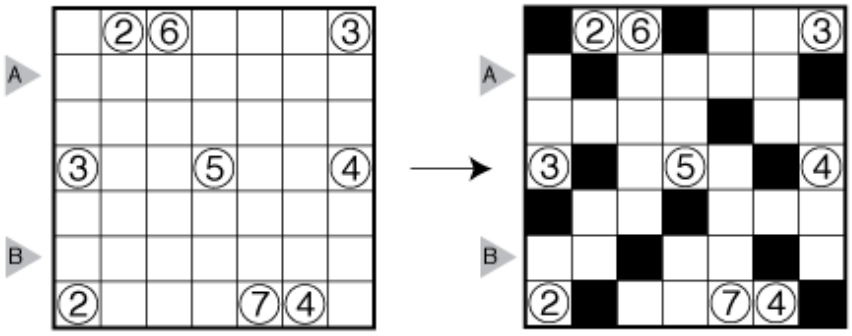
3.2 Plansza 5x5





Tym razem funkcja z punktami karnymi poradziła sobie o wiele lepiej. Dla podstawowej wersji średni czas wyniósł: 3.5706164240837097 sekund, natomiast dla rozszerzonej: 0.8856637454032898 sekund.

3.3 Plansza 7x7



Przy tak dużych rozmiarach planszy i ilościach liczb, standardowa funckja fitness nie znalazła rozwiązania ani razu, ponieważ prawie zawsze musiała zwrócić najniższy możliwy wynik, przez złamanie jednej z zasad. Funkcja z karami znajdywała rozwiązanie ze skutecznością około 40%. Średni czas z udanych prób wynosił 23.11241764068603 sekundy.

3.4 Wniosek

Jedyny przypadek, kiedy zastosowanie standardowej funkcji Fitness jest dobrym rozwiązaniem, to przy bardzo małych planszach. W reszcie przypadków, to funkcja z karami przynosi o wiele lepsze efekty, a nawet udaje jej się rozwiązać plansze, której zwykła funkcja nie jest w stanie.

4 Rój cząstek

Ponieważ funkcje Fitness przyjmują chromosomy składające się wyłącznie z 0 i 1, w roju skorzystamy z binary PSO.

Po przetestowaniu różnych ustawień algorytmu zdecydowałem się na poniższe:

```
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9, 'k': 2, 'p': 1}

optimizer = ps.discrete.BinaryPSO(n_particles=500,
dimensions=grid_height * grid_width, options=options)

optimizer.optimize(f, iters=1000, verbose=True)
```

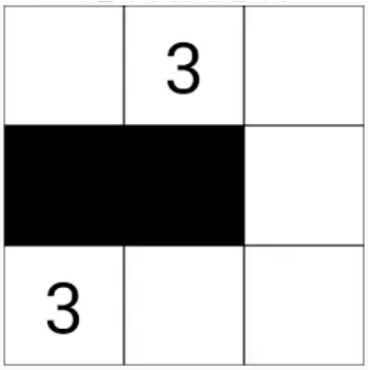
Dla funkcji f, która wygląda następująco.

```
def f(x):
    """
    x: numpy.ndarray of shape (n_particles, dimensions)

    """
    n_particles = x.shape[0]
    j = [fitness_func(x[i]) for i in range(n_particles)]
    return np.array(j)
```

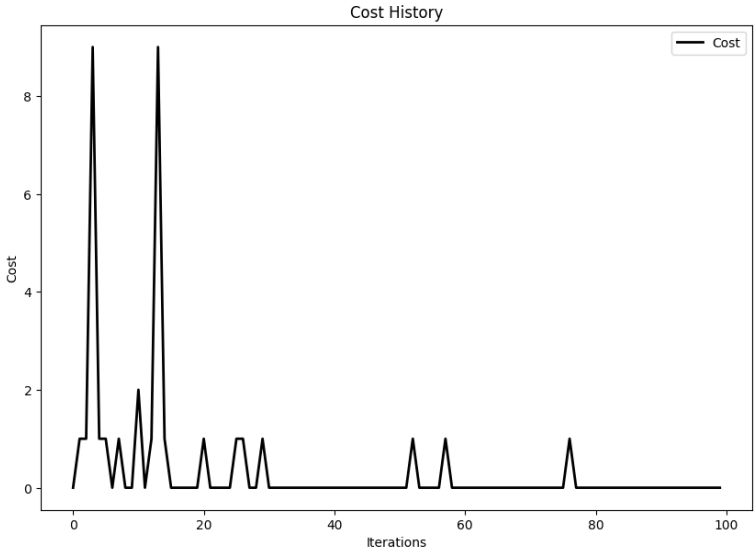
4.1 Funkcja Fitness

Ponieważ Rój wyszukuje minima, funkcje Fitness nie będą już zwracać ujemnych wartości.

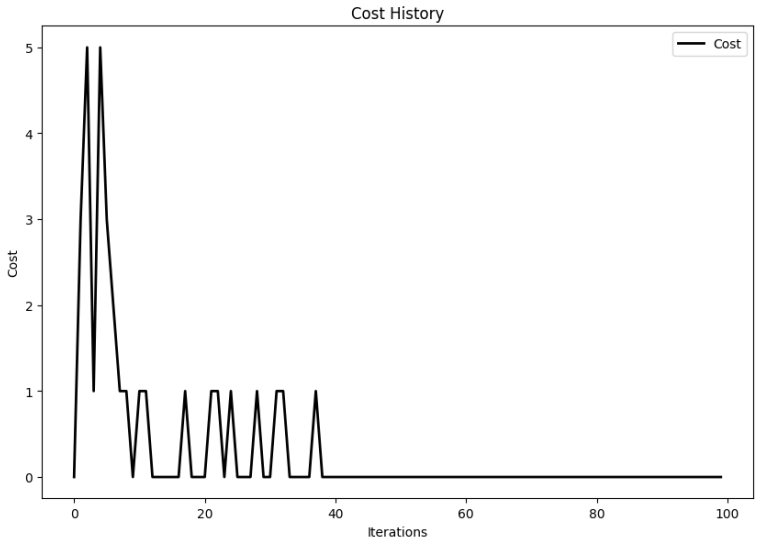
4.2 Plansza 3x3



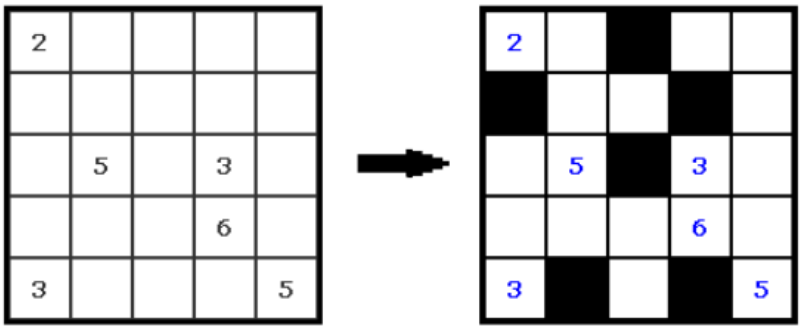
Standardowa funkcja Fitness trwała 0.4269 sekundy i nie miała problemów ze znalezieniem rozwiązania.



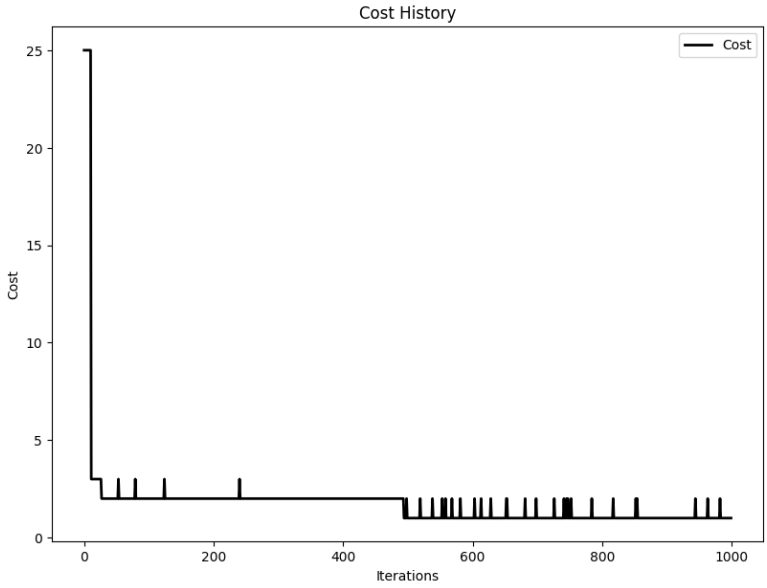
Funkcja z karami trwała 0.5400 sekundy i również poradziła sobie bezproblemowo.



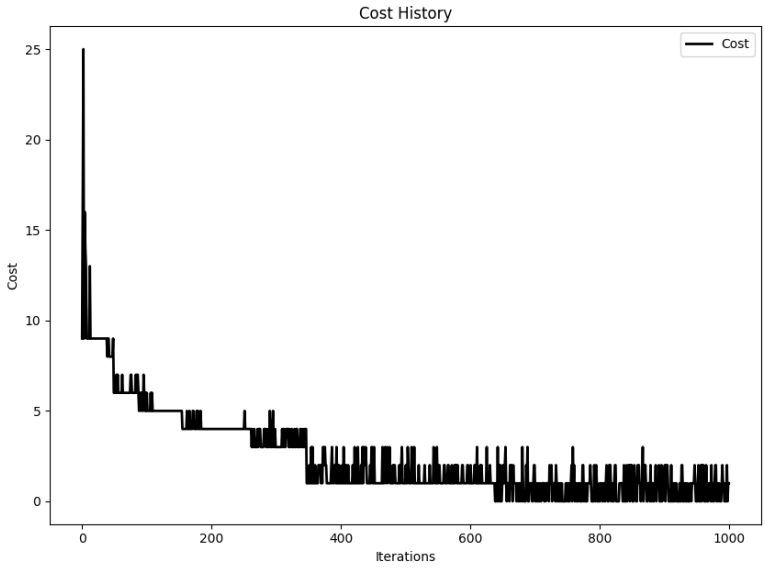
4.3 Plansza 5x5



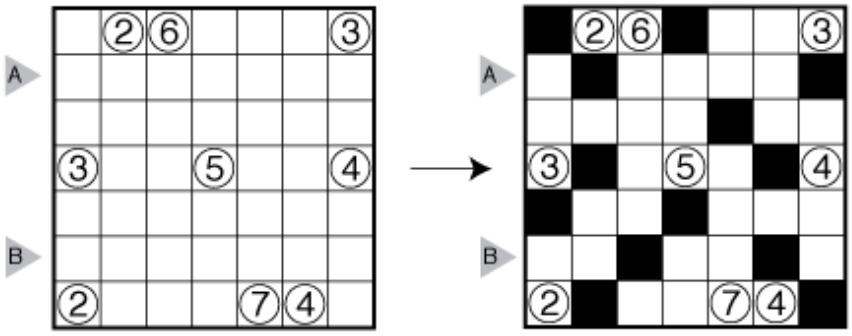
Standardowa funkcja Fitness trwała 5.7889 sekund i była w stanie jedynie zbliżyć się do rozwiązania.



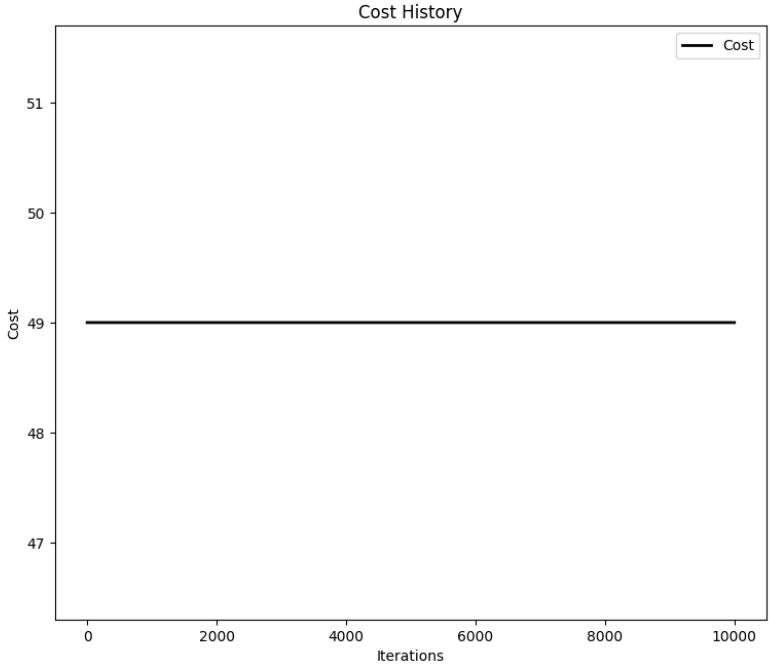
Funkcja z karami niełatwo, ale znalazła rozwiązanie. Funkcja trwała 8.2725 sekund.



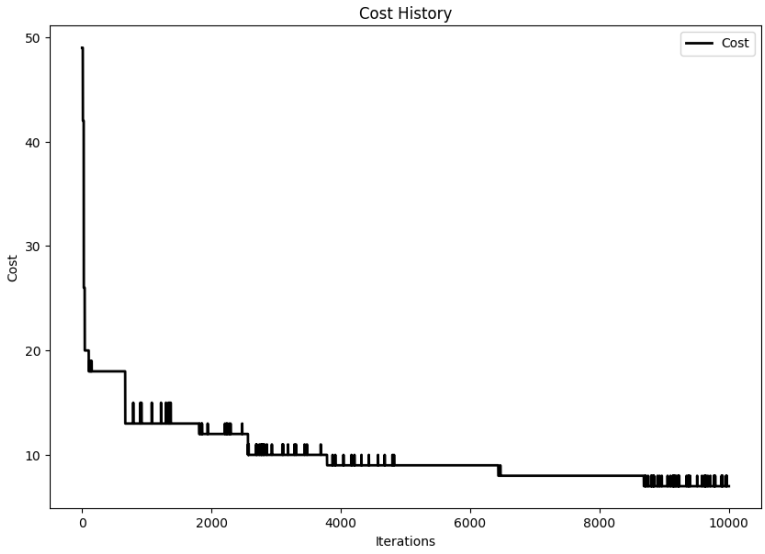
4.4 Plansza 7x7



Standardowa funkcja Fitness trwała 157.6951 sekund i nie była w stanie zwrócić jakiejkolwiek innej wartości niż najniższa możliwa.



Funkcja z karami zajęła 165.7288 sekund i nie dała rady znaleźć rozwiązania, ale była blisko.



4.5 Wnioski

W przypadku roju cząstek, funkcja Fitness z karami znowu wypadła lepiej, jednakże nawet dla niej plansza 7x7 stanowi duże wyzwanie, którego nie udaje się pokonać nawet w 10000 iteracjach.

5 Podsumowanie

Zarówno algorytm genetyczny jak i rój cząstek są w stanie znaleźć rozwiązania dla plansz 3x3 oraz 5x5. W przypadku planszy 7x7 algorytm genetyczny sprawdza się lepiej, ponieważ może nie zawsze, ale jednak znajduje rozwiązanie.

W przypadku obu algorytmów, jedynym przypadkiem kiedy warto rozważyć skorzystanie ze standardowej funkcji Fitness, jest plansza 3x3. Kiedy mamy do czynienia z większą planszą to wersja funkcji z karami będzie dawała lepsze rezultaty.

Kuromasu jest złożonym obliczeniowo problemem. Dla naszej planszy 7x7, szansa że wszystkie pola z liczbami będą białe to $\frac{1}{2^9}$, a to nie jedyna zasada, która musi zostać spełniona. Nic więc dziwnego że algorytmy mają problem ze znalezieniem rozwiązania dla większych plansz.

6 Źródła

- https://en.wikipedia.org/wiki/KuromasuSolution_methods
- https://theses.liacs.nl/pdf/20-TimvanMeurs.pdf