Projekt z Inteligencji Obliczeniowej Fill-a-pix

Cyprian Lazarowski

24.01.2023

1. Wprowadzenie

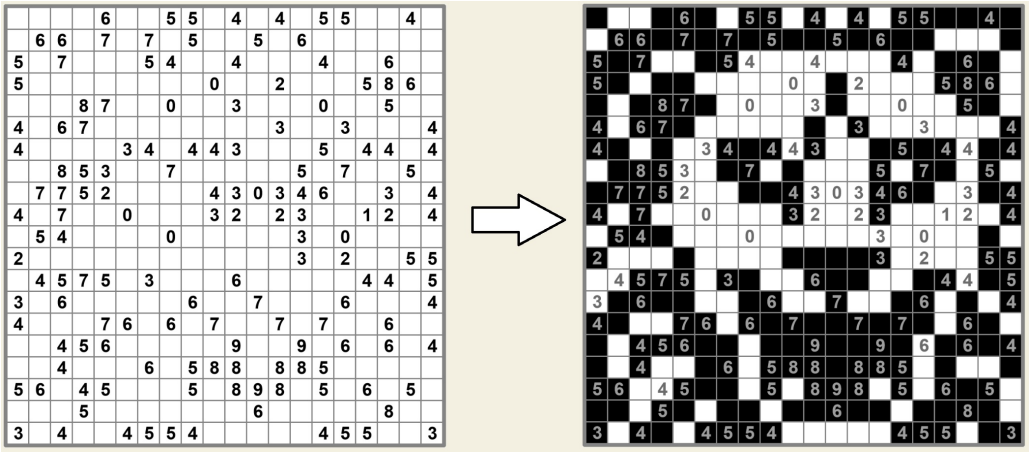
1.1. Treść zadania

Rozwiązać łamigłówkę Fill-a-pix przy pomocy algorytmu generycznego i PSO. Zbadać czas i jakość poszczególnych algorytmów, oraz przetestować go dla różnych wielkości planszy.

1.2. Opis zasad gry

Fill-a-pix jest to jedna z łamigłówek, w miarę podobnych do nonogramów. Otrzymujemy planszę podzieloną na małe kwadraty, w których mogą się znajdować liczby z przedziału (0-9), oznaczają one ilość zamalowanych kratek, będących w sąsiedztwie. Niektóre z nich są puste, nie dając nam bezpośrednich informacji, ile mają zamalowanych sąsiadów.

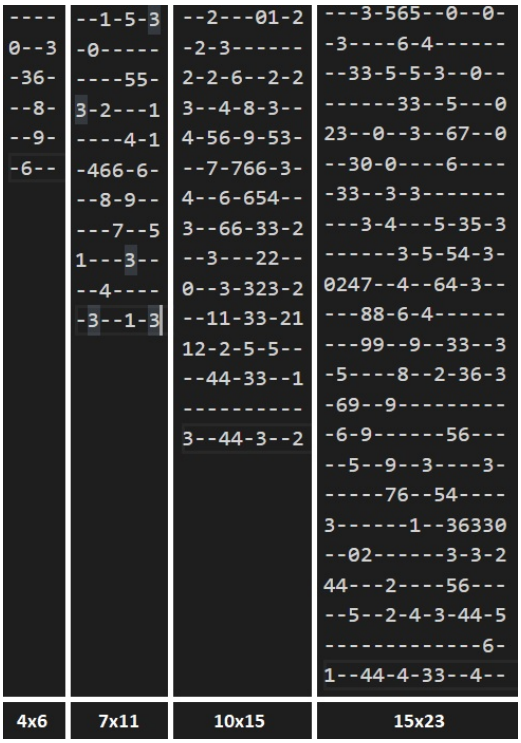
Oto przykładowa łamigłówka, o rozmiarach 20x20:



2. Implementacja

2.1. Zaimportowanie Fill-a-pixów

Użytkownik zapisuje plik z rozszerzeniem .txt, do folderu fill-a-pix. Na jego podstawie, program tworzy liste, odpowiadającą konkrentnej planszy do gry. Sposób formatu zapisywanego, do pliku wygąda następująco.



Podane wyżej przykłady, są tymi na których podstawie będziemy, próbowali uruchomić algorytmy. Będziemy sprawdzać czas i jakośc rozwiązań, dla plansz (4x6, 7x11, 10x15 i 15x23).

2.2. Algorytm genetyczny

2.2.1. Implementacja

Podstawą algorytmu genetycznego jest dobrze zaprogramowania funkcja fitness oraz odpowiedni dobór genów. Dla fill-a-pixów uznałem, że zamalowane pola będziemy oznaczać numerem jeden, natomiast puste zerem. Długość jednego rozwiązania, będzie równa liczbie kolumn pomnożonych przez ilość wierszy. Do tego zadania użyłem biblioteki pygad, która już ma zaimplementowany ten algorytm. Pozostało mi tylko wpisać odpowiednie ziemienne i napisać funkcje fitness.
Funkcja fitness natomiast została napisana na dwa sposoby:

- Obliczamy dla każdej kratki błędy i zwracamy ich liczbę w odjęciu od długości genu,
- Zwracamy liczbę poprawnych krate.

```
def fintess_func_factory(x,y,fill_a_pix,calculate_max_result,option):
    def fitness_func(solution, solution_idx):
        mistake = 0
        for i in range(y):
            for j in range(x):
                if fill_a_pix[i][j]!="-":
                    id_tab = get_all_neighborhood_id(i,j,x,y)
                    mistake+=count_score(id_tab,solution,int(fill_a_pix[i][j]),option)

        return calculate_max_result-mistake
    return fitness_func
```

Podana wyżej funkcja obsługuje oba te przypadki w zależności oczywiście od otrzymanych argumentów. Dla których:

- x - liczbę wierszy,
- y - liczbę kolumn,
- fill a pix - lista z przykładem,
- calculate max result - najlepszy wynik (dla rozwiązania w pełni poprawnego),
- option - określa typ fukcji fitness, dla której zwraca odpowiednią liczbę błędów.

Są użyte tutaj dwie funkcje pierwsza get all neighborhood id, oblicza wszystkie indexy, które są sąsiadami do aktualnego. Natomiast funkcja count score, oblicza liczbę błędów dla konkrentej kratki.

```
def count_score(id_tab, solution, num, option):
    count = 0
    for i in id_tab:
        if solution[i]==1:
            count+=1
    if option==0: return abs(num-count)
    elif option==1:
        if num == count: return 0
        else: return 1
```

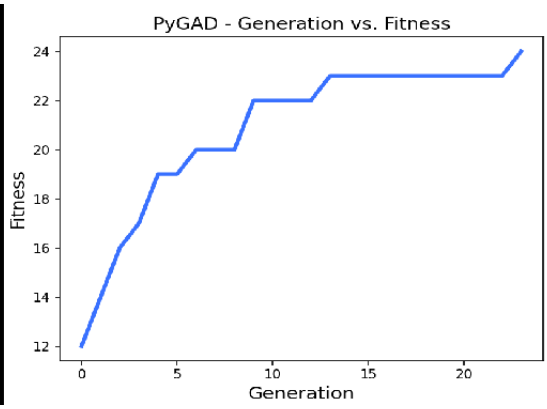
Na koniec implementacji zostały do uwzględnienia konkretne parametry dla algorytmu genetycznego. Wyglądają one następująco:

```
ga_instance = pygad.GA(gene_space=[0,1],
    num_generations=500,
    num_parents_mating=5,
    fitness_func=fintess_func_factory(x,y,fill_a_pix,max_result,option),
    sol_per_pop=20,
    num_genes=x*y,
    parent_selection_type="sss",
    keep_parents=2,
    crossover_type="single_point",
    mutation_type = "random",
    mutation_percent_genes = 10,
    stop_criteria=[f"reach_{max_result}"],
    )
```

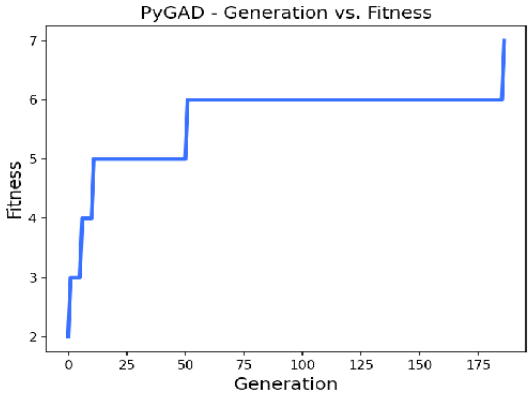
2.2.2. Porównanie wyników

Przykład 1 (4x6):

```
Genetic Algorithm (option 1):
The Best Solution:
0 0 0 0
0 0 0 1
0 0 1 1
0 1 1 1
1 1 1 1
1 1 1 1
Fitness value of the best solution = 24
Max score: 24
Time: 0.04897165298461914
```



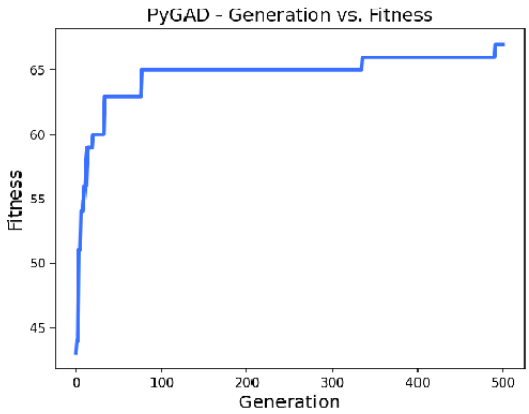

```
Genetic Algorithm (option 2):
The Best Solution:
0 0 0 0
0 0 0 1
0 0 1 1
0 1 1 1
1 1 1 1
1 1 1 1
Fitness value of the best solution = 7
Max score: 7
Time: 0.3967559337615967
```



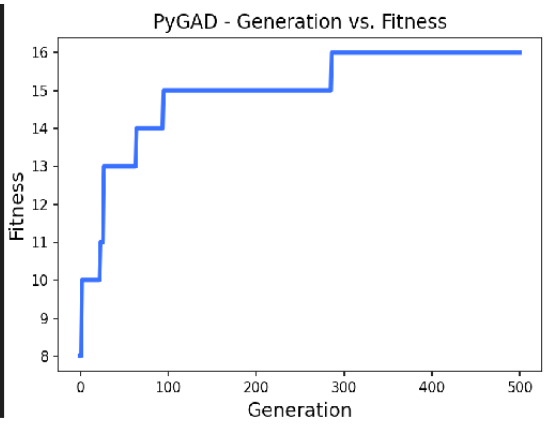
Jak można zauważyć, algorytm w tym przypadku dał radę rozwiązać dwa w miarę proste fill-a-pixy, z czego funkcja fitness oceniająca wynik na podstawie liczby niepoprawnych kratek, wykonała pracę zdecydowanie szybciej i przy miejszej liczbie iteracji.

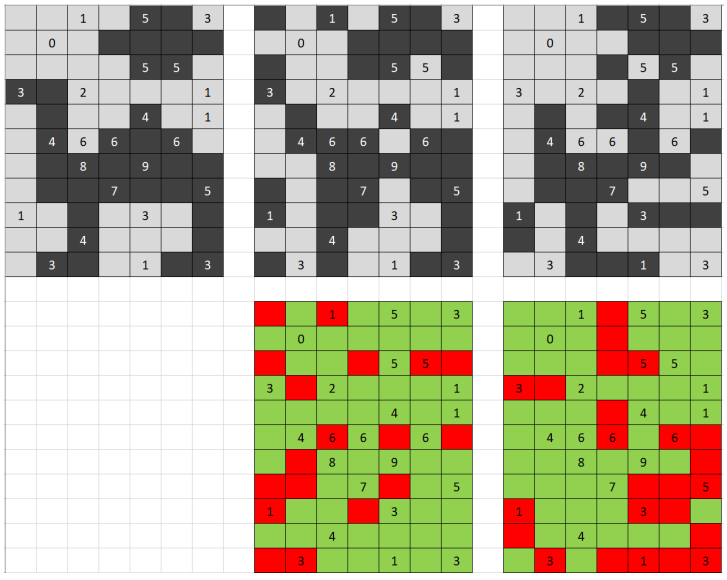
Przykład 2 (7x11):

```
Genetic Algorithm (option 1):
The Best Solution:
1 0 1 0 1 1 0
0 0 0 1 1 1 1
1 0 0 1 1 0 1
1 0 0 0 0 0 0
0 1 0 0 1 0 0
0 1 1 1 0 1 1
0 0 1 1 1 1 1
1 0 1 1 0 1 1
1 0 1 1 0 0 1
0 0 1 0 0 0 1
1 0 1 0 0 1 1
Fitness value of the best solution = 67
Max score: 77
Time: 2.0050230026245117
```




```
Genetic Algorithm (option 2):
The Best Solution:
0 0 0 1 1 1 0
0 0 0 0 1 1 1
0 0 0 1 0 1 0
0 0 0 0 1 0 0
0 1 0 1 1 0 0
0 1 0 0 1 0 1
0 1 1 1 1 1 0
0 1 1 1 0 0 0
1 0 1 0 1 1 1
1 0 1 0 0 0 0
0 0 1 1 1 0 0
Fitness value of the best solution = 16
Max score: 25
Time: 1.7951924800872803
```

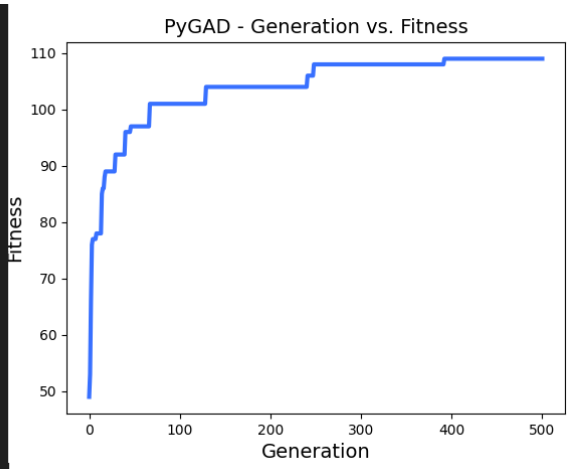




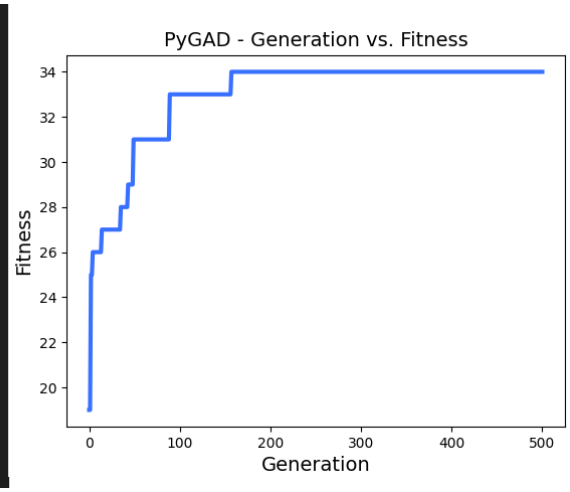
Od tego przykładu dalsze, nie będą już znajdować poprawnego wyniku, bedą starały się jak najbardziej odwzorować poprawny wynik. Dla podanych przykładów pierwszy fitness zrobił 18 błedów, natomiast opcja druga zrobiła ich już 23. Według wykresu można stwierdzić, że koło setnej iteracji algorytm, powoli się stabilizuje i dalej nie zmienia zbytnio rozwiązania.

Przykład 3 (10x15):

```
Genetic Algorithm (option 1):
The Best Solution:
0 1 0 0 0 0 0 0 1 0
0 0 0 1 1 1 0 0 1 1
0 1 0 0 1 1 0 1 1 0
1 0 0 0 1 1 0 0 0 0
1 1 1 1 1 1 0 1 1 0
1 0 0 1 1 1 1 0 1 0
0 1 1 1 0 1 0 1 1 0
1 1 0 1 1 0 0 1 0 1
0 1 0 1 0 0 0 0 0 1
0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 0 1 0 1 0
0 0 0 0 0 0 1 0 0 0
1 1 1 1 0 0 1 0 0 0
0 0 1 0 1 0 0 0 1 0
1 0 1 0 1 1 1 1 1 1
Fitness value of the best solution = 109
Max score: 150
Time: 4.1509013175964355
```

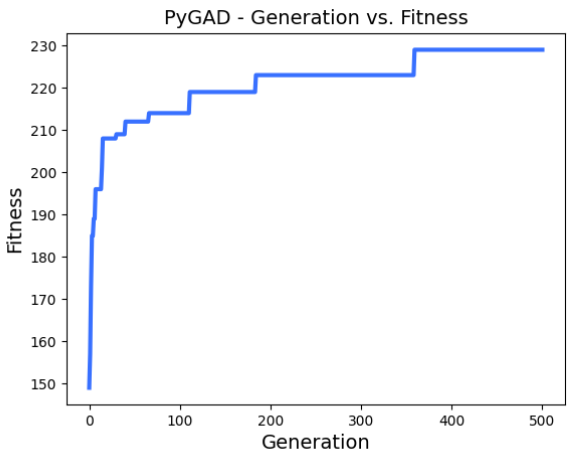



```
Genetic Algorithm (option 2):
The Best Solution:
1 1 0 1 1 0 0 1 0 1
0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 1 0 0 0 0
0 0 1 1 1 1 1 1 0 1
0 0 0 0 0 1 0 1 0 0
0 1 1 1 0 1 1 1 0 0
1 1 0 1 0 0 1 0 0 1
0 1 0 0 0 1 0 0 1 0
0 0 1 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0 0 1
0 0 1 0 1 0 1 1 0 0
1 0 0 0 1 0 0 0 0 0
0 0 0 1 1 1 1 0 0 1
1 1 0 1 0 1 1 1 0 0
1 1 1 1 1 0 0 0 1 1
Fitness value of the best solution = 34
Max score: 67
Time: 3.4778378009796143
```

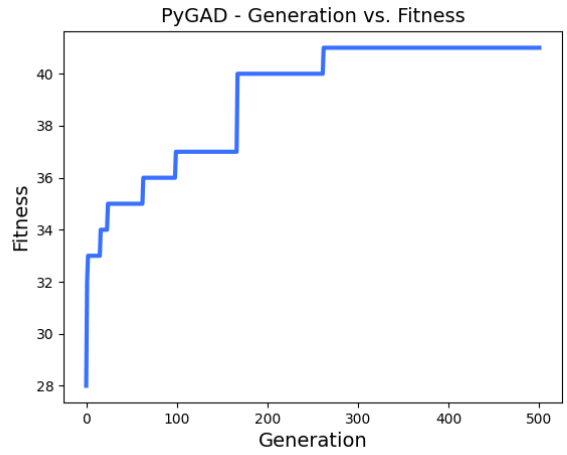


Przykład 4 (15x23):

```
Genetic Algorithm (option 1):
The Best Solution:
0 1 1 1 1 1 1 0 1 0 1 0 1 1 0
0 0 1 0 0 1 1 1 1 0 0 0 0 1 0
1 0 0 0 1 0 1 0 0 0 1 0 1 1 0
1 1 0 1 0 0 0 1 0 1 0 1 1 0 1
1 0 1 0 0 0 1 0 0 0 1 0 1 0 0
0 0 0 0 0 0 0 1 0 0 0 1 1 1 0
1 0 0 0 0 1 1 0 1 1 1 0 1 1 1
1 1 1 0 1 1 0 0 1 1 0 1 0 1 0
0 0 1 0 0 0 1 0 0 0 0 1 0 0 0
0 0 1 1 1 1 0 1 0 0 1 0 1 1 0
1 0 1 1 1 0 1 0 1 0 1 0 1 0 1
1 0 0 1 1 1 1 1 1 1 0 0 0 0 0
1 0 0 1 1 0 1 0 1 0 0 1 1 1 1
1 1 1 1 0 1 1 1 0 0 0 0 1 1 1
1 1 0 1 1 0 1 0 0 0 1 1 0 0 1
1 0 0 1 1 0 1 0 1 1 0 1 1 0 0
0 0 0 1 1 1 1 0 0 1 0 1 1 1 0
0 0 0 0 0 1 1 0 1 0 1 0 1 0 0
1 1 0 1 1 1 0 1 1 0 1 1 0 0 1
0 0 0 1 0 0 0 1 0 0 1 0 1 0 1
1 1 0 0 0 0 1 1 0 0 1 1 1 1 1
0 0 0 0 0 0 0 0 1 0 0 0 1 0 0
0 0 1 1 1 1 1 1 1 0 1 0 1 0 0
Fitness value of the best solution = 229
Max score: 345
Time: 6.933146715164185
```




```
Genetic Algorithm (option 2):
The Best Solution:
0 0 1 1 0 1 1 0 0 1 0 0 1 1 0
1 0 0 0 1 0 0 1 1 0 1 0 1 1 0
1 0 0 1 0 0 1 0 1 0 1 0 1 0 0
0 0 0 0 1 0 1 0 0 1 0 1 1 0 0
0 1 0 1 0 1 0 1 0 0 1 0 1 1 1
1 1 0 1 1 0 0 0 1 1 1 1 1 0 1
0 0 1 1 0 0 1 1 0 1 0 1 0 0 0
1 0 1 0 1 0 0 0 0 1 1 1 0 1 0
0 0 0 0 1 1 0 1 1 0 0 0 0 0 1
1 0 0 1 1 1 1 0 0 1 1 1 0 0 0
1 0 0 1 0 0 0 0 0 1 0 0 1 0 1
0 1 1 0 0 0 0 1 1 0 1 0 1 1 0
1 1 1 0 0 0 0 1 0 1 0 0 0 0 1
0 1 0 1 1 1 1 1 1 0 0 1 1 1 0
0 1 1 0 1 0 0 0 1 1 1 1 1 1 0
0 0 1 0 0 1 1 0 1 0 0 1 0 1 0
0 0 1 1 0 1 1 1 0 0 0 1 0 1 0
0 1 1 1 1 0 0 1 0 1 0 1 0 0 1
1 1 0 1 1 0 0 0 1 0 0 0 1 0 1
1 0 0 0 0 1 1 0 0 0 1 0 1 0 0
0 1 0 0 1 1 1 0 0 0 0 1 0 1 0
0 0 1 1 1 0 1 0 0 1 0 0 1 1 1
1 0 1 1 1 0 0 1 0 0 1 1 0 1 1
Fitness value of the best solution = 41
Max score: 116
Time: 6.289363145828247
```



Z tego co widać, podane parametry niezbyt sprawdzają się przy podawaniu lepszych rozwiązań. Stwierdziłem więc, że zwiększe liczbę iteracji do 10 tysięcy oraz zmienie mutację z 10 na 1. Po uruchomieniu programu, rozwiązania zdecydowanie się poprawiły, algorytm fitness opcji pierwszej dał radę rozwiązać plansze 7x11 natomiast, pozostałe wyniki były bardzo zbliżone do poprawnego wyniku.

```
Genetic Algorithm (option 1):
The Best Solution:
0 0 0 0 1 1 0
0 0 0 1 1 1 1
0 0 0 0 1 1 0
1 1 0 0 0 0 0
0 1 0 0 1 0 0
0 1 0 1 1 1 0
0 1 1 1 1 1 1
0 1 1 1 1 1 1
0 0 1 0 0 0 1
0 0 1 0 0 0 1
0 1 1 0 0 1 1
Fitness value of the best solution = 77
Max score: 77
Time: 8.72057843208313
================================
Example: ex_3.txt (10x15)
================================
Genetic Algorithm (option 1):
The Best Solution:
0 1 0 0 1 0 0 0 1 1
0 1 0 0 1 1 0 0 0 0
0 0 0 0 1 1 0 0 1 0
0 1 0 0 1 1 1 0 0 1
1 1 0 1 1 1 1 0 0 0
0 1 0 1 1 1 1 1 1 0
0 1 1 1 0 0 0 1 0 0
1 1 0 1 1 1 0 0 0 1
0 0 0 1 1 0 0 1 1 0
0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 1 0 1 0 1
0 1 0 0 0 0 1 0 0 0
0 0 1 0 1 0 1 1 1 0
1 0 1 1 0 0 0 0 0 0
1 1 0 1 1 1 1 1 1 1
Fitness value of the best solution = 145
Max score: 150
Time: 58.72235560417175
```

```
Genetic Algorithm (option 2):
The Best Solution:
0 0 0 0 1 1 0
0 0 0 1 1 1 1
0 0 0 0 1 0 1
1 1 1 1 0 0 0
0 1 1 1 0 0 0
0 0 0 1 0 1 0
1 0 1 1 1 1 1
0 0 1 0 0 1 1
0 0 1 1 1 0 1
1 0 1 0 0 0 1
0 0 1 0 0 1 1
Fitness value of the best solution = 21
Max score: 25
Time: 29.011703968048096
================================

================================
Genetic Algorithm (option 2):
The Best Solution:
0 1 0 0 0 0 0 0 1 1
1 0 0 1 0 0 0 0 0 0
0 0 0 1 1 1 0 0 0 1
1 0 0 0 1 1 1 0 1 0
1 1 1 1 1 1 1 0 0 0
0 1 1 0 1 1 1 1 0 1
1 1 1 0 1 1 0 0 1 0
0 1 1 0 1 0 0 1 0 0
0 0 0 1 1 1 0 0 1 0
0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 1 0 1 0
1 0 0 0 0 0 1 1 0 0
0 0 1 0 0 0 1 0 0 0
0 1 1 1 1 0 0 0 1 0
1 1 0 0 1 1 1 1 1 0
Fitness value of the best solution = 63
Max score: 67
Time: 62.45916223526001
```

Z lewej strony pierwsza opcja fitness, z prawej opcja numer dwa. Nadal widać zdecydowaną różnice, w działaniu jednego i drugiego. U góry wykonywany był przykład nr. 2 na dole przykład nr. 3. Na tych wszystkich zmianach, cierpi tylko czas wykonywania zadania, gdzie policzenie konkretnego przykładu o rozmiarach 10x15 wymaga od nas już blisko minuty czekania.

2.2.3. Sprawdzenie większej liczby rozwiązań

Aktualne dane, opierają się głównie na pojedyńczych uruchomieniach programu, stwierdziłem więc że dla przykładu pierwszego (4x6) jak i drugiego (7x11) uruchomie blisko sto testów, aby pozyskać srednią, oraz sprawdzić czy będzie dawało radę ze znajdywaniem rozwiązań. Wyszły mi następujące wyniki:

Dla opcji nr. 1 z planszą 4x6:

```
Program time: 0.01099395751953125, Solution: 24
Program time: 0.016989946365356445, Solution: 24
Program time: 0.019989013671875, Solution: 24
Program time: 0.009994983673095703, Solution: 24
Program time: 0.030980348587036133, Solution: 24
Program time: 0.021987438201904297, Solution: 24
Program time: 0.04397273063659668, Solution: 24
Program time: 0.012991189956665039, Solution: 24
Program time: 0.018987655639648438, Solution: 24
Program time: 0.017994165420532227, Solution: 24
Program time: 0.12991905212402344, Solution: 24
Program time: 0.026980876922607422, Solution: 24
Program time: 0.016989469528198242, Solution: 24
Program time: 0.025985002517700195, Solution: 24
Program time: 0.10788631439208984, Solution: 24
Program time: 0.019986629486083984, Solution: 24
Program time: 0.19488096237182617, Solution: 24
Program time: 0.02298450469970703, Solution: 24
Program time: 0.02198624610900879, Solution: 24
Program time: 0.014990806579589844, Solution: 24
Program time: 0.04297280311584473, Solution: 24
Program time: 0.016993045806884766, Solution: 24
Program time: 0.042977333068847656, Solution: 24
Program time: 0.04297351837158203, Solution: 24
Program time: 0.024990081787109375, Solution: 24
All: 100, Find: 100, Not find: 0, Mean: 24
Time Mean: 0.04943981885910034, Max: 0.28882265090942383, Min: 0.009994983673095703
```

Dla opcji nr. 2 z planszą 4x6:

```
Program time: 8.138645648956299, Solution: 7
Program time: 0.7319505214691162, Solution: 7
Program time: 2.0577914714813232, Solution: 7
Program time: 1.4298045635223389, Solution: 7
Program time: 0.10906648635864258, Solution: 7
Program time: 0.17949485778808594, Solution: 7
Program time: 4.607417345046997, Solution: 7
Program time: 11.644942283630371, Solution: 7
Program time: 0.30346179008483887, Solution: 7
Program time: 1.624640703201294, Solution: 7
Program time: 1.1895089149475098, Solution: 7
Program time: 1.1002070903778076, Solution: 7
Program time: 1.0496044158935547, Solution: 7
Program time: 0.2604222297668457, Solution: 7
Program time: 6.9754650592803955, Solution: 7
Program time: 0.1959242820739746, Solution: 7
Program time: 0.06997275352478027, Solution: 7
Program time: 2.30906772613525 4, Solution: 7
Program time: 0.31299901008605957, Solution: 7
Program time: 4.884620904922485, Solution: 7
Program time: 0.06591057777404785, Solution: 7
Program time: 0.056524038314819336, Solution: 7
Program time: 0.22567033767700195, Solution: 7
All: 100, Find: 100, Not find: 0, Mean: 7
Time Mean: 1.556465849876404, Max: 15.308583974838257, Min: 0.016707897186279297
```

Porównując oba wyniki, jesteśmy w stanie stwierdzić że przy aktualnych parametrach algorytmu plansza 4x6 powinna być prawie zawsze rozwiązywana. Czasowo jednak, widać znaczą przewagę funkcji fitness opierającej się na liczeniu różnicy zamalowanych kratek, względem numeru określającego dany kwadrat.

Dla opcji nr. 1 z planszą 7x11:

```
Program time: 0.8274924755096436, Solution: 77
Program time: 26.833478927612305, Solution: 76
Program time: 26.46970272064209, Solution: 76
Program time: 16.50185680389404 3, Solution: 77
Program time: 27.57202386856079, Solution: 76
Program time: 1.9108245372772217, Solution: 77
Program time: 27.51805567741394, Solution: 76
Program time: 1.0993225574493408, Solution: 77
Program time: 12.41735553741455, Solution: 77
Program time: 27.31618642807007, Solution: 76
Program time: 1.2212481498718262, Solution: 77
Program time: 27.419116973876953, Solution: 76
Program time: 26.952406644821167, Solution: 75
Program time: 26.640597105026245, Solution: 76
Program time: 26.541661024093628, Solution: 76
Program time: 27.415120601654053, Solution: 76
Program time: 14.657975435256958, Solution: 77
Program time: 2.1007065773010254, Solution: 77
All: 100, Find: 60, Not find: 40, Mean: 76
Time Mean: 15.173076744079589, Max: 30.633157968521118, Min: 0.30281519889831543
```

Dla opcji nr. 2 z planszą 7x11:

```
Program time: 31.453347206115723, Solution: 24
Program time: 5.175461053848267, Solution: 25
Program time: 32.22075629234314, Solution: 23
Program time: 31.456220626831055, Solution: 23
Program time: 30.998340845108032, Solution: 24
Program time: 32.34435224533081, Solution: 23
Program time: 29.327059030532837, Solution: 24
Program time: 28.708325147628784, Solution: 22
Program time: 28.05672574043274, Solution: 23
Program time: 28.651358127593994, Solution: 22
Program time: 28.841246843338013, Solution: 23
Program time: 31.423582792282104, Solution: 22
Program time: 31.39984965324402, Solution: 24
Program time: 30.64812731742859, Solution: 24
Program time: 27.514075994491577, Solution: 22
Program time: 27.284201622009277, Solution: 23
All: 100, Find: 15, Not find: 85, Mean: 23
Time Mean: 25.73521169424057, Max: 32.34435224533081, Min: 2.179659605026245
```

Tutaj sytuacja nieco bardziej staje się ciekawsza. Można zauważyć, że 40 w pierwszym i 85 w drugim, odpowiedzi jest niepoprawnych. Jednak patrząc po średniej, można wywnioskować, że wartości są w miare zbliżone do głównego wyniku. Widać nadal sporą różnicę między dwoma różnymi podejściami fitness. Zgodzić się można, który jest dokładniejszy, a co za tym idzie szybszy (przez wzgląd na obecność kryterium końcowego).

2.3. Particle Swarm Optimizer (PSO)

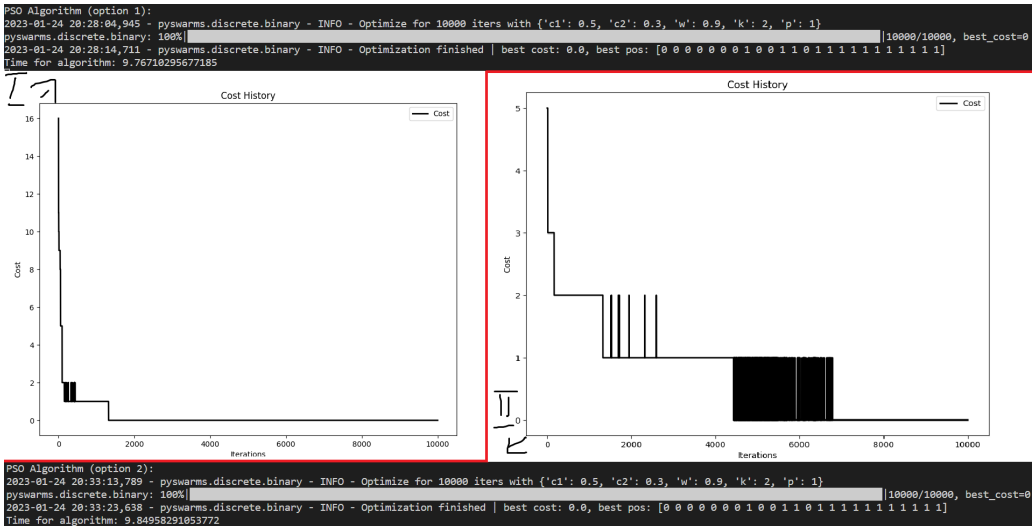
2.3.2. Implementacja

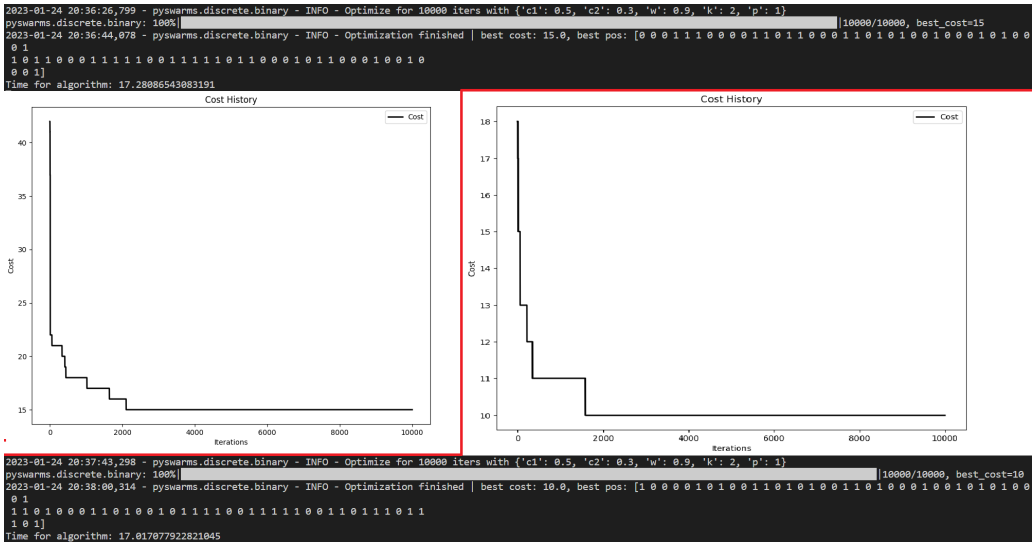
Do rozwiązywania zadań przy pomocy PSO wykorzystałem bibliotekę pyswarm. Do tego zadania użyłem BinaryPSO z następującymi parametrami:

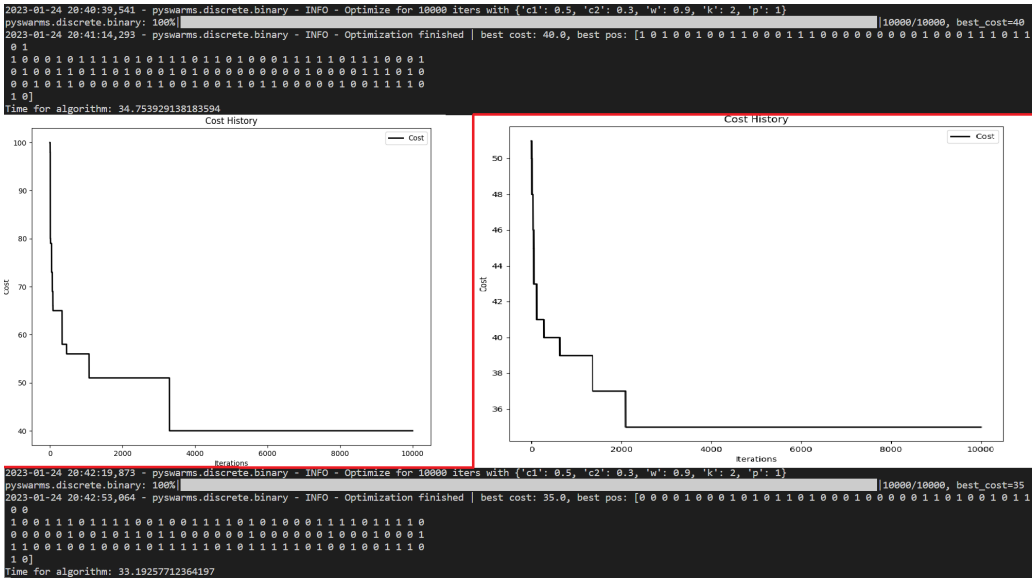
```
options = {'c1': 0.5, 'c2': 0.3, 'w':0.9, 'k':2, 'p':1}
kwargs = {'z': x, 'y': y, 'fill_a_pix': fill_a_pix, 'max': max_resoult, 'option': option}
optimizer = ps.discrete.BinaryPSO(n_particles=10, dimensions=x*y, options=options)
optimizer.optimize(fitness_PSO, iters=50000, verbose=True, **kwargs)
```

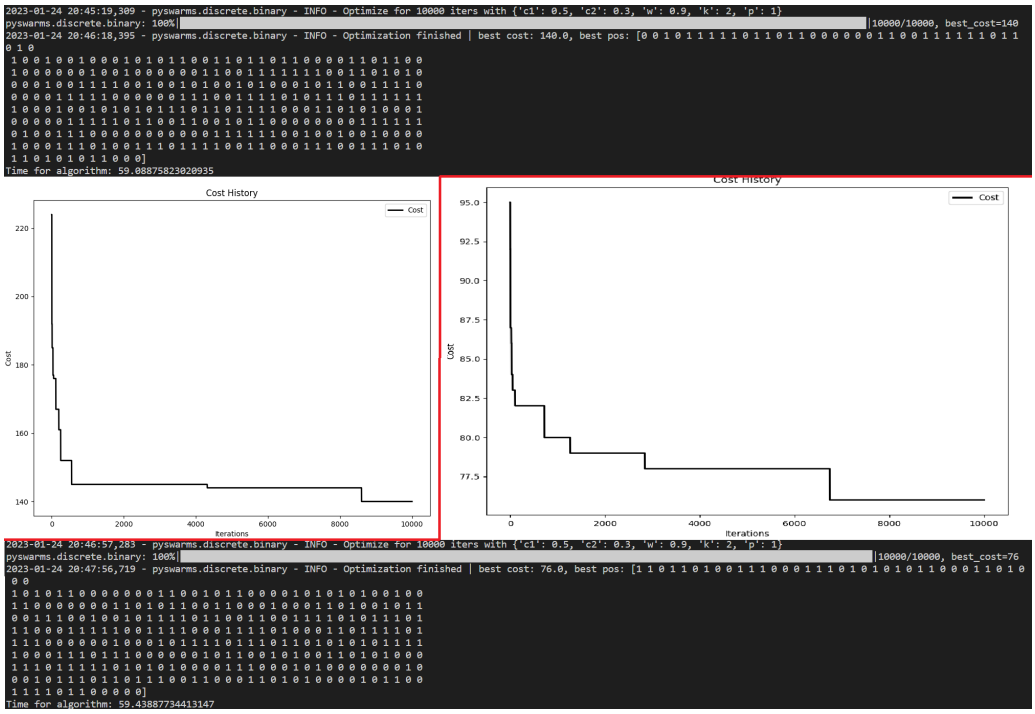
Nauczony poprzednim eksperymentem, z góry zwiększyłem liczbę iterazji do 10 tysięcy. Funkcje fitness natomiast lekko zmodyfikowałem, usuwając informacje o maksymalnej liczbie w rozwiązaniu. Aktualnie zwraca ona liczbę wykrytych błędów. Tak jak poprzednio badam dwa sposoby obliczania funkcji fitness.

2.3.3. Wyniki









Wyniki zostały przedstawione dla poprzednich czterech przykładów. Jak można zauważyć, PSO zaczyna mieć problemy już przy drugim z nich podobnie jak to było przy starych ustawieniach algorytmu genetycznego. Ze względu na brak czasu, pozostanę przy jednym przykładzie, na podstawie którego wysnuję hipotezę, że algorytm genetyczny jest lepszy w rozwiązywaniu tego typu problemów.