Rozwiązywanie Fill-a-pixów

1. Informacje o problemie

Fill-a-pix to łamigłówka graficzna podobna w pewnym stopniu do nonogramów. Mamy planszę (zazwyczaj kwadratową), podzieloną na małe kwadraty, na której należy pomalować niektóre z nich na czarno w taki sposób, aby wszystkie zamalowane kratki utworzyły spójny obrazek. O zasadzie zamalowania kratek informuje nas odpowiednia liczba w pisana w danym kwadracie która mówi, ile kratek wokół danego kwadratu (łącznie z nim samym) ma być zamalowana.

Problemem, który należało rozwiązać było znalezienie dobrego algorytmu, który rozwiązywałby ww. fill-a-pix, zaimplementowanie takiego algorytmu, przetestowanie jego działania oraz czasu jego działania.

2. Rozwiązanie problemu

2.1 Przygotowania

Na samym początku, żeby rozwiązać mój problem, potrzebowałem jakieś fill-a-pixy, na których mógłbym testować mój algorytm. Dodałem więc 3 różnej wielkości „mozaiki” przerobione tak, by program mógł je odczytywać.

```
board_large = [
    [2,-1,3,-1,3,-1,3,-1,4,5,5,4,-1,-1,0],
    [-1,-1,-1,3,-1,4,-1,4,-1,-1,-1,-1,-1,-1,-1],
    [1,-1,2,1,2,2,3,2,2,2,-1,-1,4,3,-1],
    [-1,-1,3,-1,3,-1,3,-1,0,-1,-1,-1,-1,4,-1],
    [-1,-1,-1,-1,-1,-1,3,-1,-1,-1,-1,5,7,-1,-1],
    [5,-1,-1,7,-1,6,-1,-1,-1,2,-1,-1,-1,-1,-1],
    [-1,-1,5,-1,7,5,5,3,-1,-1,-1,-1,5,4,1],
    [5,-1,-1,7,-1,7,-1,-1,-1,-1,-1,7,-1,-1,-1],
    [-1,-1,7,-1,-1,8,9,-1,-1,9,-1,-1,9,6,-1],
    [5,-1,7,-1,8,-1,7,-1,9,-1,-1,-1,8,7,-1],
    [4,-1,-1,-1,6,-1,-1,-1,6,6,-1,-1,-1,7,5],
    [-1,-1,6,-1,-1,5,6,-1,-1,-1,-1,-1,-1,-1,-1],
    [-1,8,7,-1,-1,7,-1,5,-1,5,8,8,-1,-1,4],
    [-1,8,7,-1,-1,-1,-1,-1,4,-1,8,-1,5,-1,2],
    [-1,-1,-1,-1,3,-1,5,-1,-1,-1,-1,-1,-1,-1,-1]
    ]
```

1. Dodany w kodzie fill-a-pix o rozmiarze 15x15

W miejscach, gdzie postawiona jest -1, interpreter odczytuje daną kratkę jako pustą. Jeżeli jest jednak wstawiona liczba 0-9, jest to odczytywane jako wymagana liczba zamalowanych kratek wokół.

2.2 Implementacja algorytmu - genetyczny

Mając coś na czym mogę testować algorytm, przystąpiłem do wstępnej implementacji. Jako możliwe wartości rozwiązań, ustawiłem 0 i 1, gdzie 0 jest równe zamalowanej kratce, a 1 niezamalowanej (jest to odwrócone ustawienie, ze względu na późniejsze wyświetlanie rozwiązania algorytmu, gdzie 0 – czarna kratka, a 1 – biała kratka).

Zasada działania programu jest dosyć prosta: algorytm genetyczny generuje rozwiązanie, które przerabiam na tablicę dwuwymiarową, niejako „nakładając" je na dany szablon z fill-a-pixem. Następnie sprawdzam wartości znajdujące się w szablonie. Jeżeli program wykryje liczbę w przedziale od 0 do 9, przechodzi do zliczania czarnych kratek w rozwiązaniu, na tej samej pozycji w tablicy, w której znaleziona była liczba w szablonie. Funkcja zliczająca liczbę zamalowanych kratek wokół jest najobszerniejszą w programie, ponieważ zaimplementowałem ręcznie każdy możliwy przypadek (liczba może znajdować się w środku szablonu, na danych rogach lub jego brzegach. Wtedy też zmienia się maksymalna liczba możliwych zamalowanych kratek wokół – równomiernie jest to 9, 4 i 6).

Kiedy program obliczy już liczbę zamalowanych wokół kratek, porównuje je z liczbą w szablonie i zwraca wartość bezwzględną różnicy tych dwóch liczb. Liczba ta jest odejmowana od wartości fitness (docelowo 0). Jeżeli końcowo zwracana przez funkcję fitness jest równa zeru, algorytm znalazł bezbłędne rozwiązanie.

```python
#genetic
def fitness_func(solution, solution_idx):
    solution_2d = np.reshape(solution, (len(input),len(input)))
    index_Y = 0
    fitness = 0
    for y in input:
        index_X = 0
        for x in y:
            if x in range(0,10):
                wrong_squares = check_around(index_Y, index_X, solution_2d, x)
                fitness -= wrong_squares
            index_X+=1
        index_Y+=1
    return fitness
```

2. Funkcja fitness, na podstawie której wiemy czy dane rozwiązanie jest prawidłowe.

2.3 Implementacja algorytmu – optymalizacja przez rój

Funkcja fitness dla optymalizacji przez rój wygląda podobnie jak dla algorytmu genetycznego. Jedyna różnica jest taka, że wartość zwracana przez funkcję sprawdzającą zamalowane kwadraty jest dodawana do domyślnej wartości 0, a nie odejmowana. Jest to spowodowane tym, że algorytm roju szuka wartości minimalnej wśród rozwiązań, a genetyczny przeciwnie – maksymalnej.

```
def fitness_func_swarm(solution):
    solution_2d = np.reshape(solution, (len(input),len(input)))
    index_Y = 0
    fitness = 0
    for y in input:
        index_X = 0
        for x in y:
            if x in range(0,10):
                wrong_squares = check_around(index_Y, index_X, solution_2d, x)
                fitness += wrong_squares
            index_X+=1
        index_Y+=1
    return fitness
```

3. Funkcja fitness dla algorytmu optymalizacji przez rój.

3. Eksperymenty

3.1 Mały rozmiar (5x5)

Po zaimplementowaniu przystąpiłem do testowania działania programu. Zacząłem od najmniejszego szablonu. Po uruchomieniu programu parę razy, niemal za każdym razem algorytmy (zarówno genetyczny jak i rój) zwracały wartość 0. Oznaczało to, że program działa dla małego szablonu.

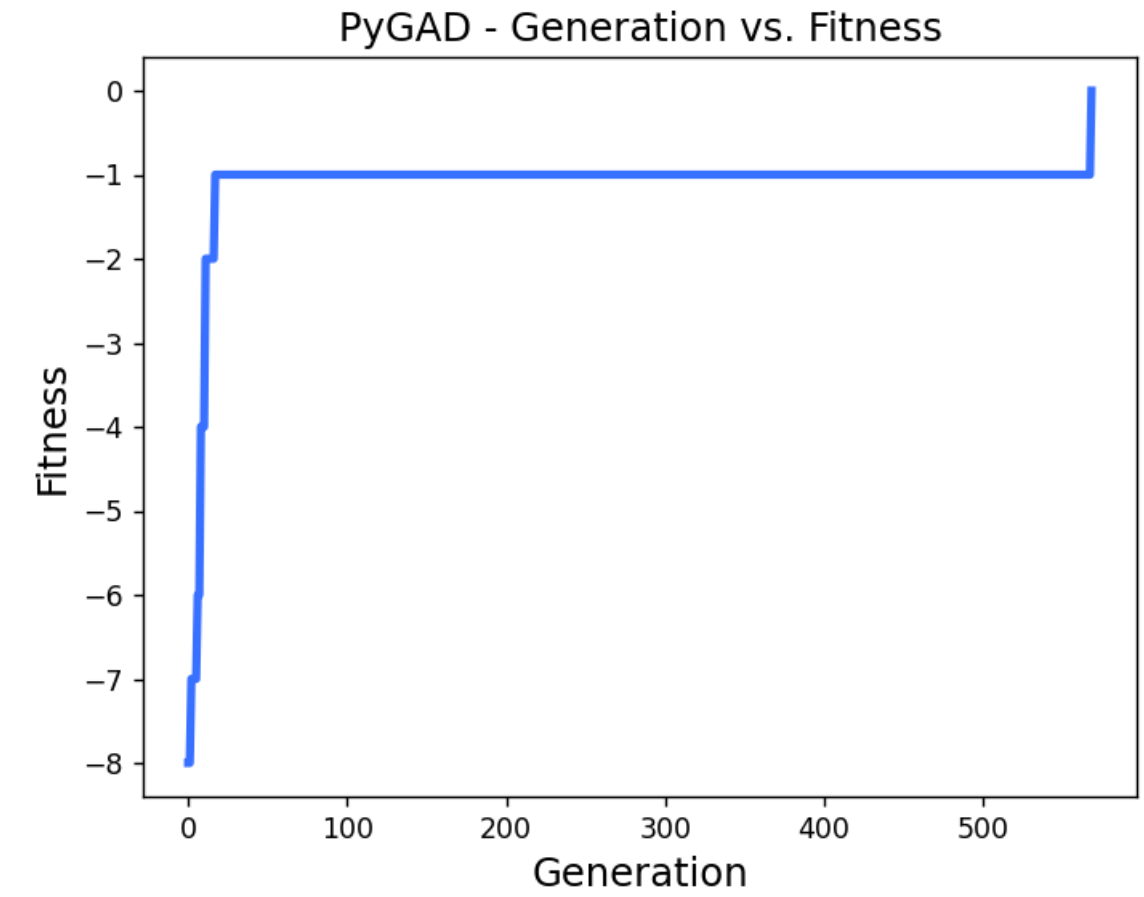
Warto jest zauważyć, że algorytm genetyczny znajdywał rozwiązanie zdecydowanie szybciej niż algorytm roju. Jest to spowodowane ustawionym w algorytmie genetycznym kryterium stopu – algorytm zatrzymuje się, gdy rozwiązanie zwraca wartość zero. Takiego ustawienia nie ma w algorytmie roju, co oznacza, że nawet jeśli algorytm znajdzie rozwiązanie, to nie zatrzyma się do czasu skończenia wszystkich iteracji.

```
sol_per_pop = 15
num_genes = len(input) * len(input)

num_parents_mating = 7
num_generations = 10000
keep_parents = 3

parent_selection_type="sss"
crossover_type="single_point"
mutation_type="random"
mutation_percent_genes = 100 / num_genes
```

4. Parametry testowe algorytmu genetycznego.



5. Przykładowy wykres wyników algorytmu genetycznego.

```
boundary=len(input)*len(input)

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

def f(x):
    return list(map(fitness_func_swarm, x))

optimizer = ps.discrete.BinaryPSO(n_particles=50, dimensions=boundary,
options=options)
start = time.time()
print("hello")
cost, pos= optimizer.optimize(f, iters=5000, verbose=True)
```

6. Parametry algorytmu roju dla małego szablonu.

	1	2	3	4	5	6	7	8	9	10	**Average**	How many times solved	
genetic	30,797	1,072	7,329	12,889	0,627	0,236	0,097	1,718	29,105	20,437	10,431		
genetic_value	-2	0	0	0	0	0	0	0	0	0			9
swarm(50 particles, 5000 iters)	43,465	43,066	43,829	47,866	43,518	43,059	43,092	43,323	43,089	43,130	43,744		
swart_cost	1	0	0	0	0	0	1	0	0	0			8

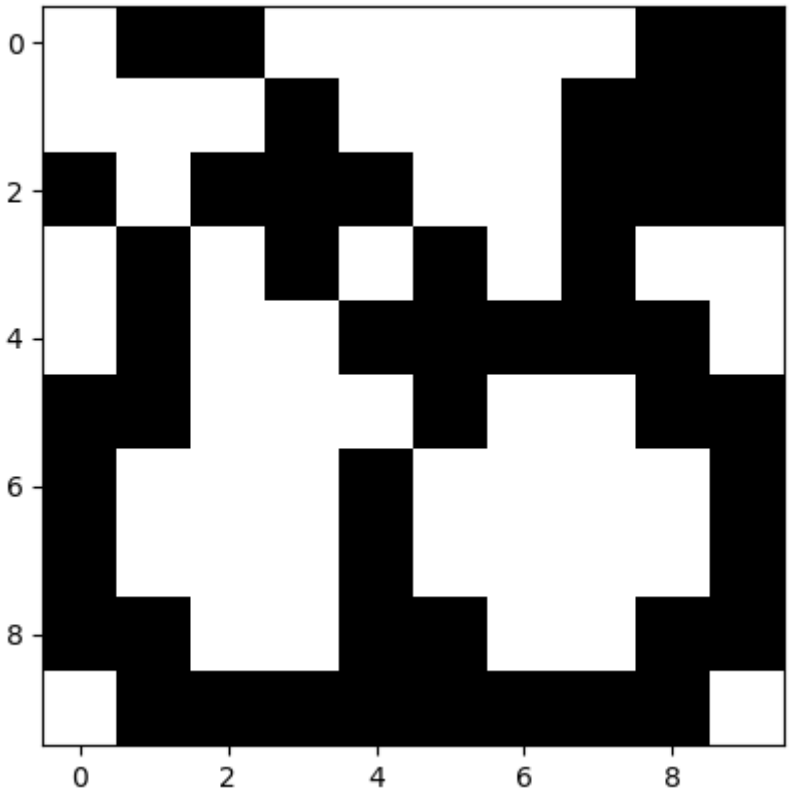
Dzięki przetestowanym czasom i rozwiązaniom da się już zauważyć pewne zależności. Pierwszą jest to, że w przypadku małego rozmiaru szablonu, oba algorytmy znajdują rozwiązania bez problemu. Drugą zależnością jest czas. Algorytm genetyczny, przez możliwość skończenia działania zaraz po wyszukaniu rozwiązania, działa o wiele szybciej. Jednak nawet jeśli nie znajdzie rozwiązania, to nadal wykonuje się szybciej niż algorytm roju, mimo że nieznacznie. Obserwacje co do czasu będą się potwierdzały w dalszej części, przy testowaniu większych szablonów.

3.2 Średni rozmiar (10x10)

	1	2	3	4	5	6	**Average**
genetic	66,431	66,078	69,086	20,216	65,794	72,000	59,934
genetic_value	-4	-2	-1	0	-4	-1	-2
swarm(50 particles, 5000 iters)	114,018	110,257	115,651	117,476	115,066		114,494
swart_cost	23	21	20	22	21		21,4
swarm(200 particles, 1000 iters)		84,017	86,132	85,375	85,213		85,184
swart_cost		23	21	22	25		22,75
swarm(500 particles, 250 iters)		53,796	54,644	54,251	54,203		54,224
swart_cost		26	30	24	23		25,75
swarm(100 particles, 20000 iters)						876,663	
						19	

Przy średnim rozmiarze zacząłem dostrzegać problemy z algorytmami. Co pierwsze rzuca się w oczy, to że oba algorytmy prawie w ogóle nie poradziły sobie z wyszukaniem rozwiązania. Na 6 przypadków tylko raz udało się to algorytmowi genetycznemu (z takimi samymi parametrami co w przypadku małego szablonu). Byłem również zdziwiony, gdy zobaczyłem tak wysoki - w porównaniu do algorytmu genetycznego – wynik algorytmu roju. Postanowiłem więc

również przetestować go z innymi parametrami – niestety bezskutecznie, wynik prawie w ogóle się nie zmienił. Doszedłem wtedy do wniosku, że algorytm roju jest beznadziejny dla naszego problemu, szczególnie biorąc pod uwagę to, że znajdował wyższe wartości w wyższym od algorytmu genetycznego czasie (rozpatrując przypadek, gdzie oba algorytmy nie znajdowały rozwiązania – dla algorytmu genetycznego średni czas działania był 67 sekund, gdzie dla roju było to zależne od ilości iteracji. Mimo tego nawet, gdy algorytm roju wykonywał się ponad dwa razy dłużej, znajdował o wiele gorsze rozwiązania. Należy też wziąć pod uwagę, że gdyby zmienić parametry genetycznego, to również mógłby dłużej się wykonywać, nawet od algorytmu roju – co nie zmienia faktu, że zwraca rozwiązania zawsze o wiele lepsze od drugiego algorytmu).

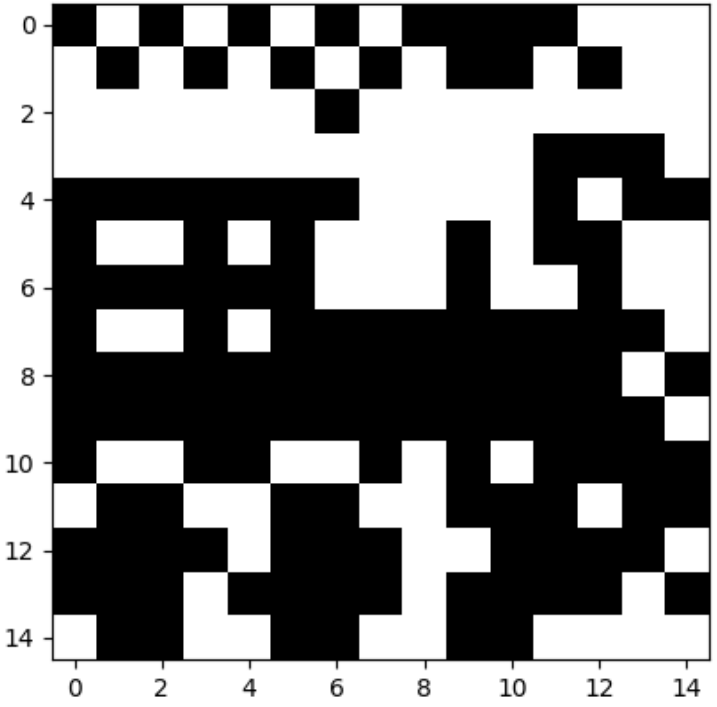


7. Rozwiązanie średniego fill-a-pixa z fitness=-1.

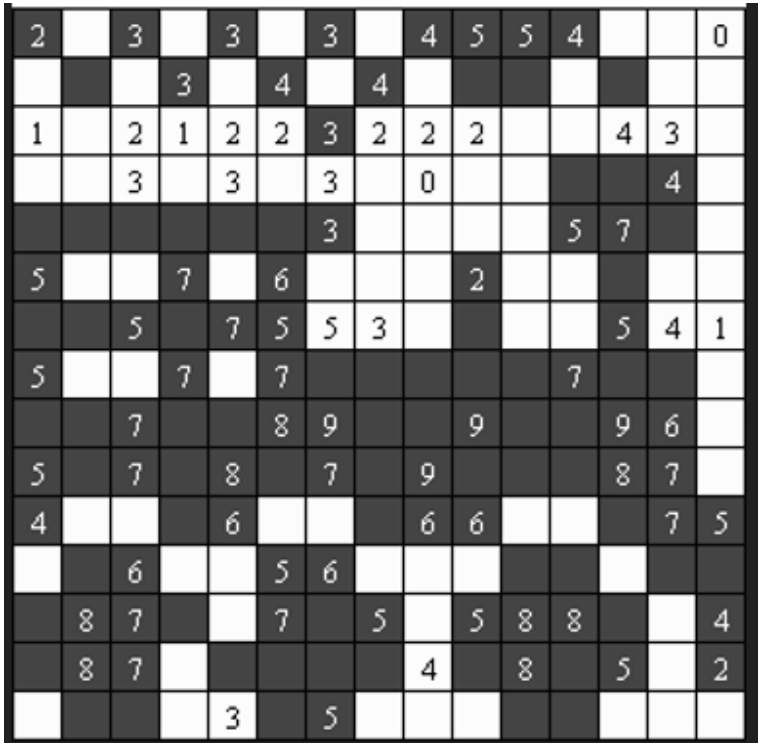
3.3 Duży rozmiar (15x15)

	1	2	3	4	5	**Average**
genetic	133,567	133,469	133,222	138,179	136,276	134,942
genetic_value	-12	-9	-9	-8	-8	-9,2
swarm	220,424	226,735	532,979	237,603	235,346	290,617
swart_cost	77	87	82	78	85	81,8

Wyniki testu dużego fill-a-fixa tylko potwierdzają testy małego i średniego. Szansa na rozwiązanie fill-a-pixa spada tym bardziej, czym większy rozmiar szablonu, a algorytm z optymalizacją roju nie nadaje się do rozwiązywania naszego problem.



8. Rozwiązanie dużego fill-a-pixa dla fitness=-4.



9. Prawidłowe rozwiązanie dużego fill-a-pixa.

4. Podsumowanie

Reasumując wszystkie aspekty naszego problemu oraz wyników testowania jego rozwiązania, wyciągnąłem następujące wnioski:

- Oba algorytmy znajdują rozwiązanie dla małej planszy
- Algorytm z optymalizacją roju nie nadaje się do rozwiązania naszego problemu
- Algorytm genetyczny jest w stanie znaleźć rozwiązanie problemu, przy odpowiednich parametrach, jednak czym większy rozmiar planszy, tym zdecydowanie mniejsza szansa na znalezienie rozwiązania.