Projekt1 Adam Pilarski
Fill-a-pix

1)Wstęp

Wybrany problem to Fill-a-pix. Problem został rozwiązany przy użyciu algorytmu genetycznego. Testy zostały przeprowadzone na 3 różnych planszach o 3 różnych rozmiarach.

2)Funkcja fitness

```
def findnotzeroindexes(board):
    flatboard=[num for sublist in board for num in sublist]
    indexes=[]
    for i in range(len(flatboard)):
        if flatboard[i]!=-1:
            indexes.append([int((i-(i%len(board[0])))/len(board[0])),i%len(board[0])])
    return indexes

indexes=findnotzeroindexes(Plansza)
```

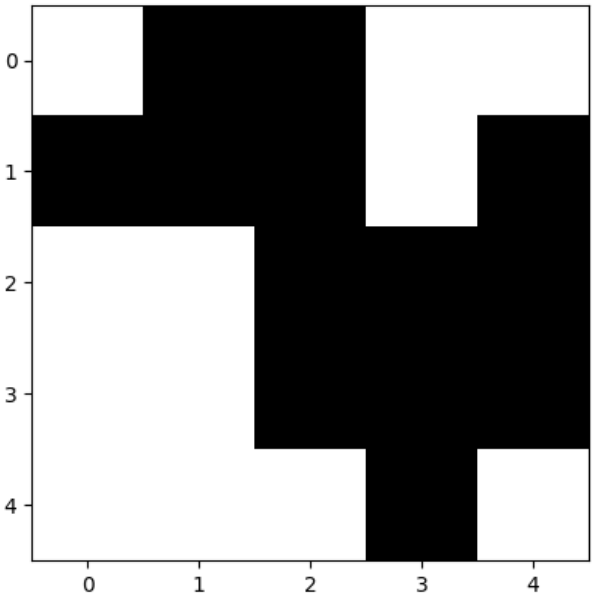
```
def make_a_board(solution,board):
    newSolution=[]
    for i in range(1,len(board)+1):
        newSolution.append(solution[(i-1)*len(board[0]):i*len(board[0])])
    return newSolution

def roundsum(board,idY,idX):
    idxY=idY+1
    idxX=idX+1
    newBoard=[[0]*(len(board[0])+2)]
    for i in range(len(board)):
        newBoard.append(board[i].tolist())
        newBoard[i+1].append(0)
        newBoard[i+1].insert(0,0)
    newBoard.append([0]*(len(board[0])+2))
    suma=0
    indexList=[[idxY-1,idxX-1],[idxY-1,idxX],[idxY-1,idxX+1],[idxY,idxX-1],
    for idx in indexList:
        if newBoard[idx[0]][idx[1]]==1:
            suma+=1
    return suma


def fitness_func(solution,solution_idx):
    sol = make_a_board(solution, Plansza)
    score=0
    for el in indexes:
        num=roundsum(sol,el[0],el[1])
        if num!=Plansza[el[0]][el[1]]:
            score-=numpy.abs(num-Plansza[el[0]][el[1]])
    return score
```

Funkcja findnotzeroindexes szuka wszystkich indeksów, na których są liczby do wypełnienia i które powinny być sprawdzone.
Funkcja make_a_board przerabia tablice jednowymiarową z wygenerowanego solution na macierz pasującą do danej planszy.
Funkcja roundsum sprawdza ile pól wokół danego indeksu jest “zamalowana”.
Funkcja fitness_func używa roundsum i findnotzeroindexes by porównać ile pól jest zamalowanych i ile powinno być, po czym odejmuje różnice od zwracanej liczby punktów.

3)Testy

Parametry stałe, które się nie zmieniały podczas testów:
sol_per_pop = 100
parent_selection_type = "sss"
crossover_type = "single_point"
mutation_type = "random"

3.1)Ptak 5x5

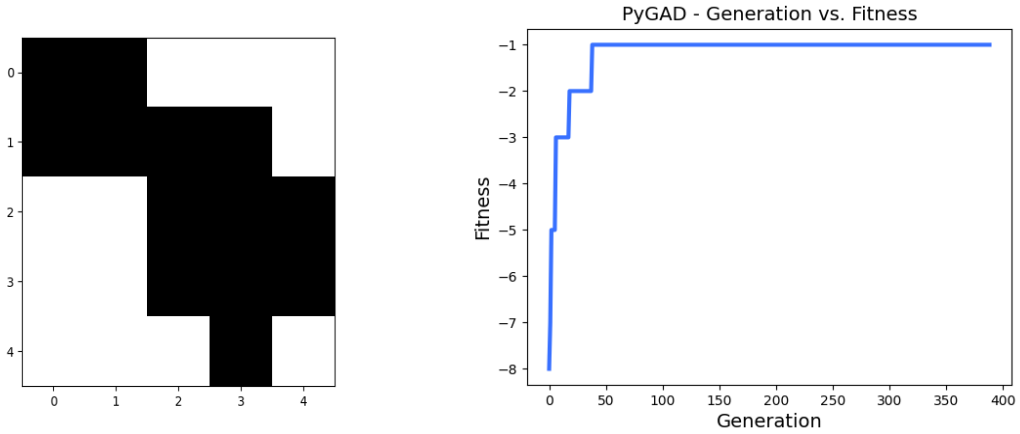


Poprawnie rozwiązana plansza

Używając odpowiednich parametrów tak mała plansza jest rozwiązana bez błędu w mniej niż 20 generacji. Parametr, który zmieniał najwięcej

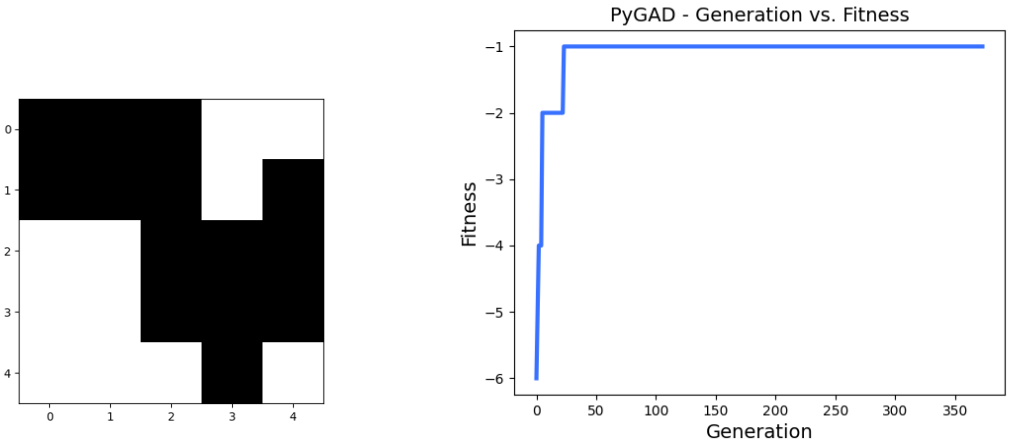
jest parametr mutation_percent_genes. W każdym przypadku parametr **keep_parents=20** a **num_parents_mating=35.**

mutation_percent_genes=20



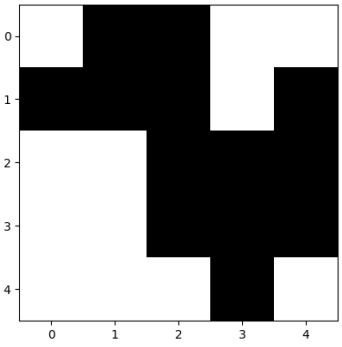
W tym przypadku po 400 generacjach rozwiązanie się zatrzymało przez parametr **saturate_350.** Rozwiązanie mimo tylu generacji jest błędne, wynik to -1, co oznacza, że jedno pole się nie zgadza.

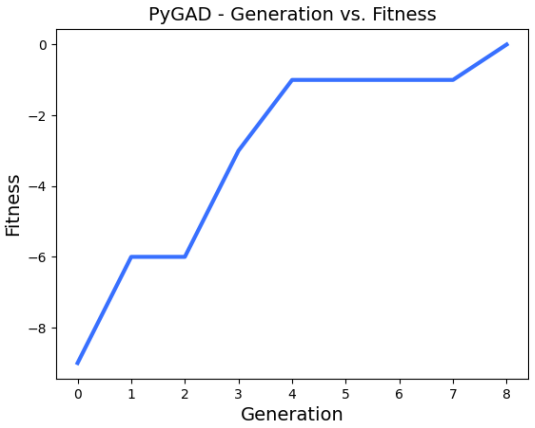
mutation_percent_genes=12



W tym przypadku również podobne wyniki. Koniec ok 400 generacji i błędne rozwiązanie z wynikiem -1.

mutation_percent_genes=5





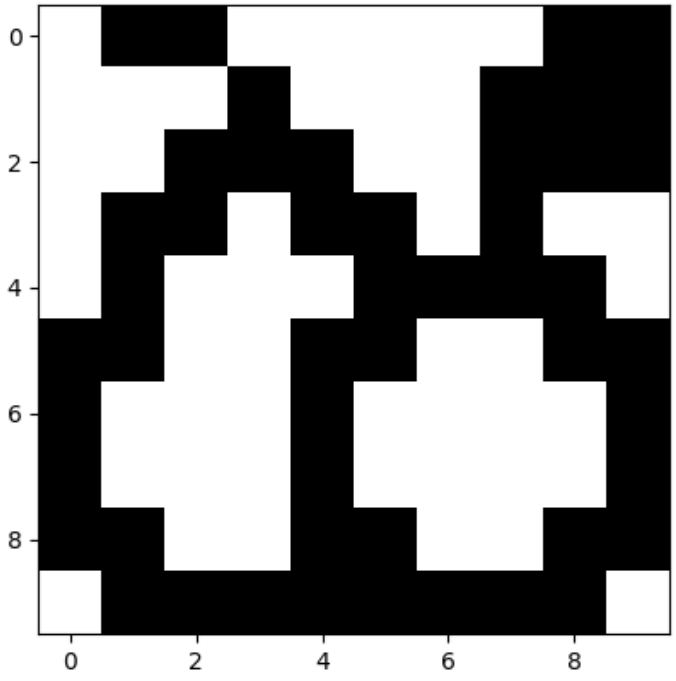
W tym przypadku poprawne rozwiązanie za każdym razem zostaje znalezione bardzo szybko.

Testy czasowe

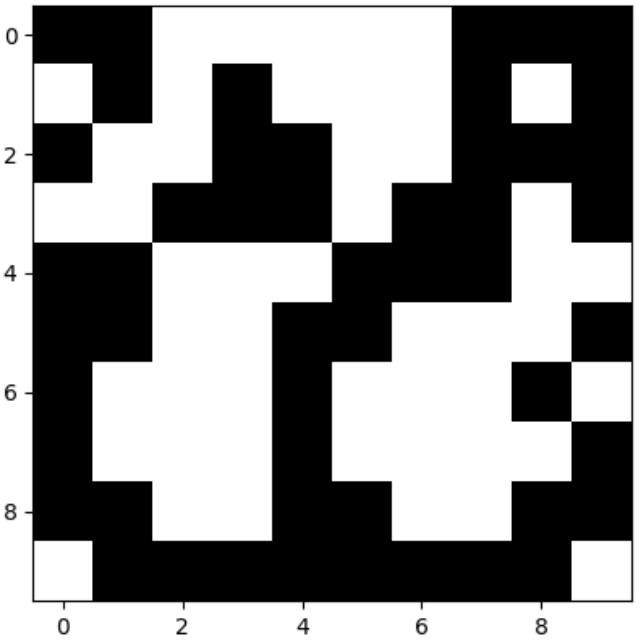
Średni czas wykonania dla 100 prób.

mutation_percent_genes=20	0.46473214864730833
mutation_percent_genes=12	0.05108705997467041
mutation_percent_genes=5	0.03794173002243042

3.2)Gruszki 10x10



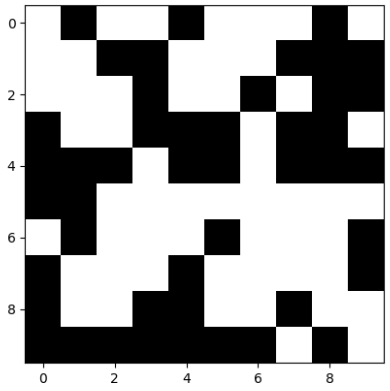
Poprawnie rozwiązana plansza

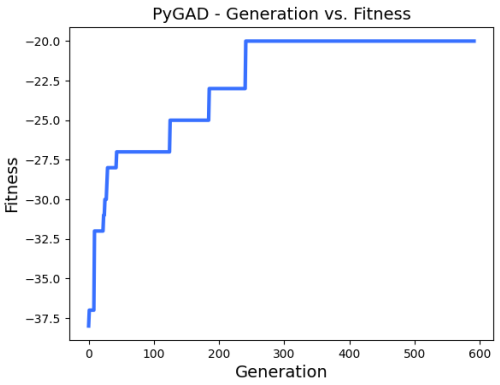


Najlepszy otrzymany wynik

Najlepszy wynik jaki udało mi się otrzymać dla tej planszy to -4 osiągnięty po 2000 generacji przy parametrach sol_per_pop = 200 oraz mutation_percent_genes = 5.

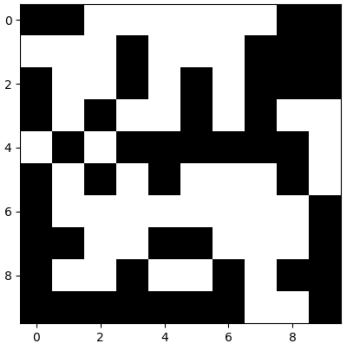
mutation_percent_genes=20
num_parents_mating=35
keep_parents=20

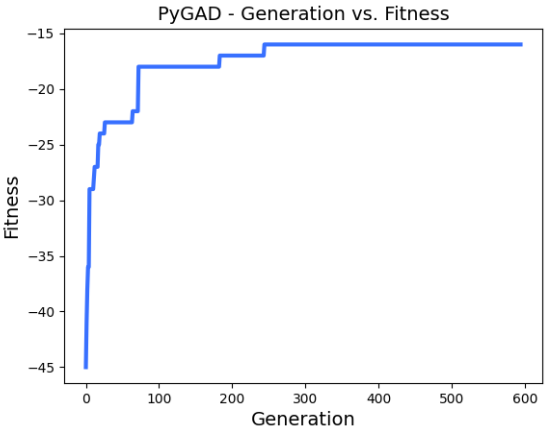




W tym przypadku duży błąd, wynik na poziomie -20, po 600 generacjach

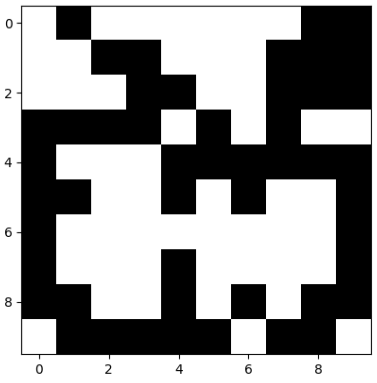
mutation_percent_genes=12
num_parents_mating=35
keep_parents=20

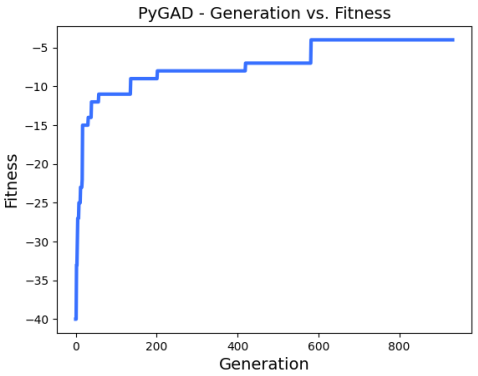




Rozwiązanie bardzo podobny jak w poprzednim przypadku, wynik -16 ok 600 generacji.

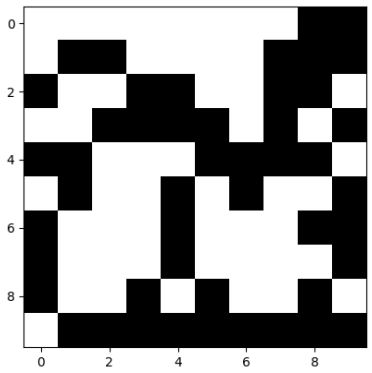
mutation_percent_genes=5
num_parents_mating=35
keep_parents=20

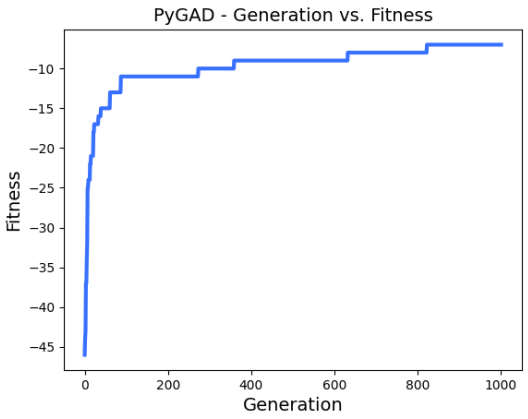




Dużo lepsze rozwiązanie z wynikiem -4, co pokazuje, że mały procent mutacji sprzyja lepszemu rozwiązywaniu tego problemu.

mutation_percent_genes=5
num_parents_mating=45
keep_parents=30



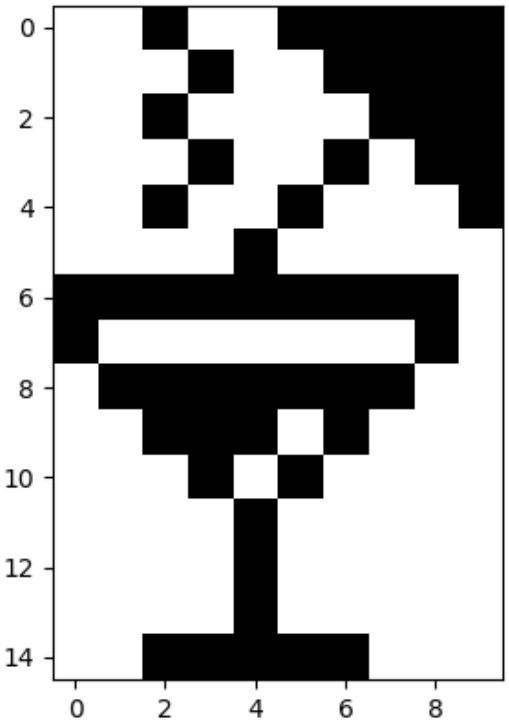


Zmiana parametrów dotyczących rozmnażania pogorszyła wyniki do wyniku -7 w 1000 generacji

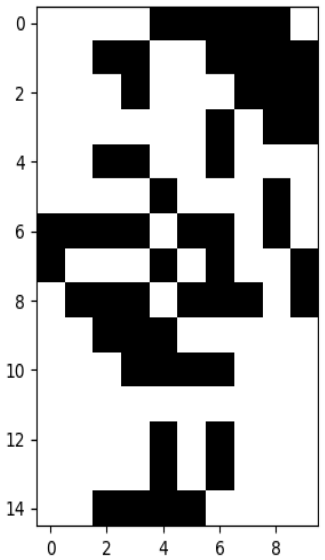
Testy czasowe
Średni czas wykonania dla 10 prób.

mutation_percent_genes=20	**19.548275303840636**
mutation_percent_genes=12	**16.74300467967987**
mutation_percent_genes=5	**26.761790585517883**

3.3)Szampan 10x15



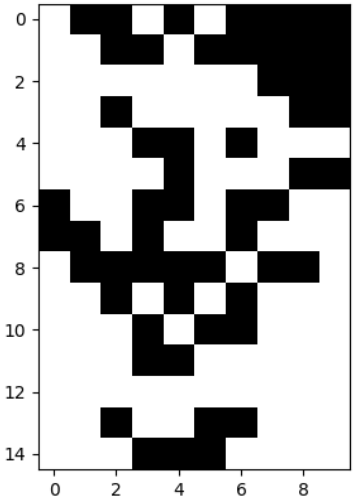
Poprawnie rozwiązana plansza

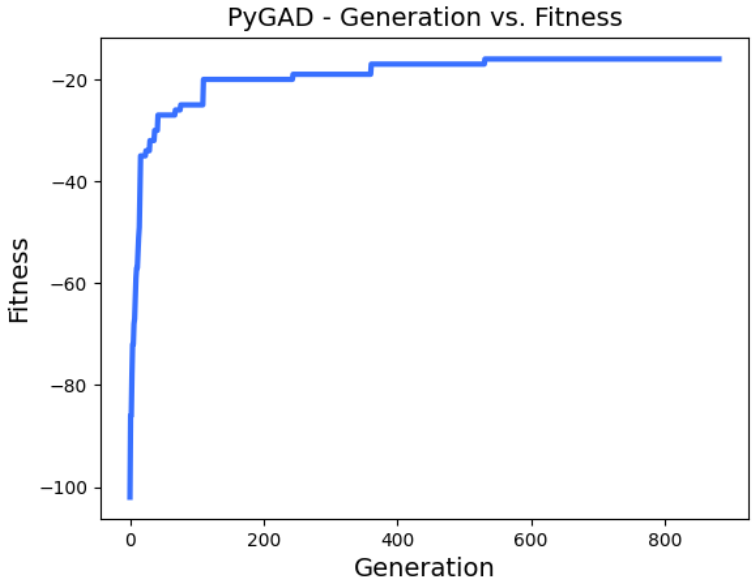


Najlepszy otrzymany wynik

Najlepszy wynik jaki udało mi się otrzymać dla tej planszy to -13 osiągnięty po 1000 generacji przy parametrach sol_per_pop = 300 oraz mutation_percent_genes = 5.

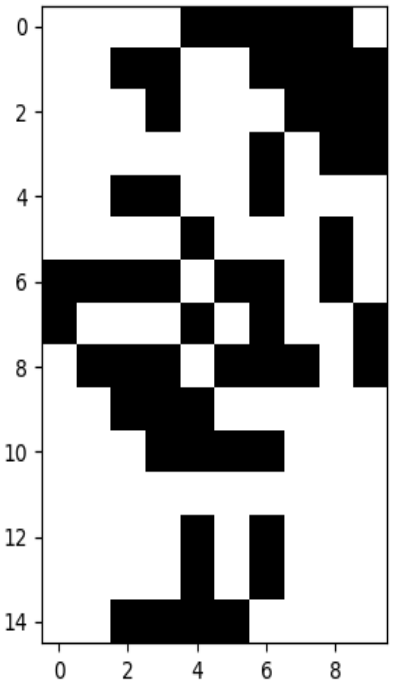
mutation_percent_genes=5
num_parents_mating=35
keep_parents=20
sol_per_pop = 100
num_generations = 1000

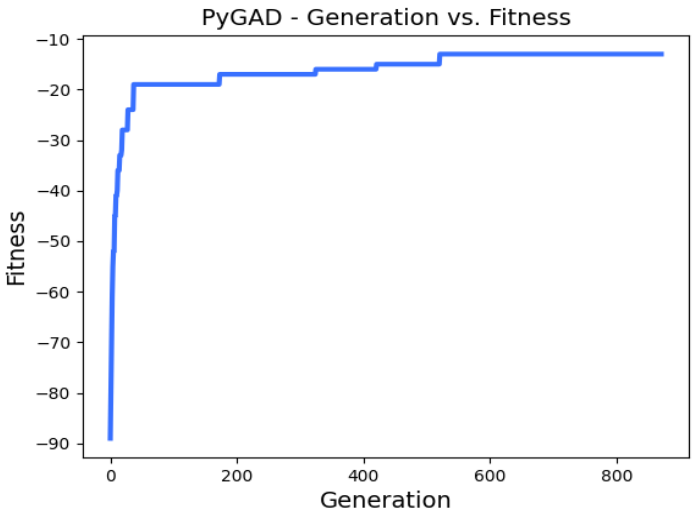




Wynik w tym przypadku to -16 przy około 900 generacjach.

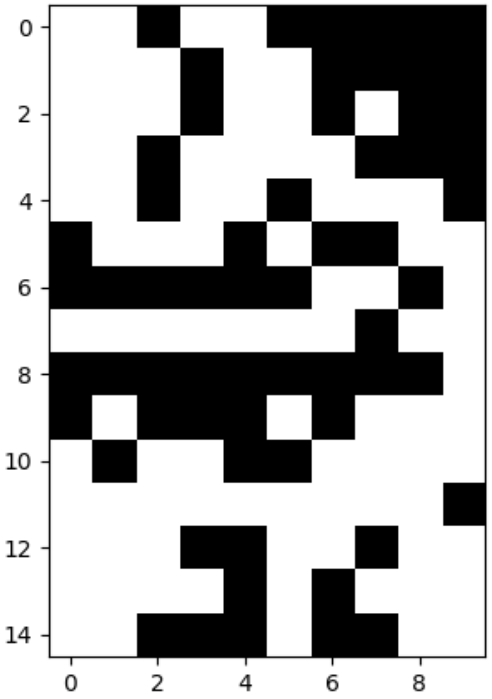
mutation_percent_genes=5
num_parents_mating=35
keep_parents=20
sol_per_pop = 300
num_generations = 1000





W tym przypadku wynik -13 przy około 900 generacjach.

mutation_percent_genes=5
num_parents_mating=35
keep_parents=20
sol_per_pop = 200
num_generations = 2000



Wynik -16, przy ponad 1400 generacji.
Testy pokazują, że przy większej ilości chromosomów dostajemy lepsze wyniki

Testy czasowe
Średni czas wykonania dla 10 prób(liczba generacji = 1000).

sol_per_pop = 100	24.184646892547608
sol_per_pop = 200	64.89672718048095
sol_per_pop = 300	93.46463465690613

4)Podsumowanie

Moje rozwiązanie tego problemu nie znajduje najlepszych rozwiązań w przypadku większych plansz.
Znalezione najlepsze parametry:
parent_selection_type = "sss"
crossover_type = "single_point"
mutation_type = "random"
mutation_percent_genes=5
num_parents_mating=35
keep_parents=20
Testy pokazują również że zwiększenie liczby chromosomów w generacji pozwala uzyskać lepsze wyniki, jednak czas potrzebny na wykonanie algorytmu również znacząco rośnie.