message = "A3FZ9YBQ" * 100 # 6400 BITOW

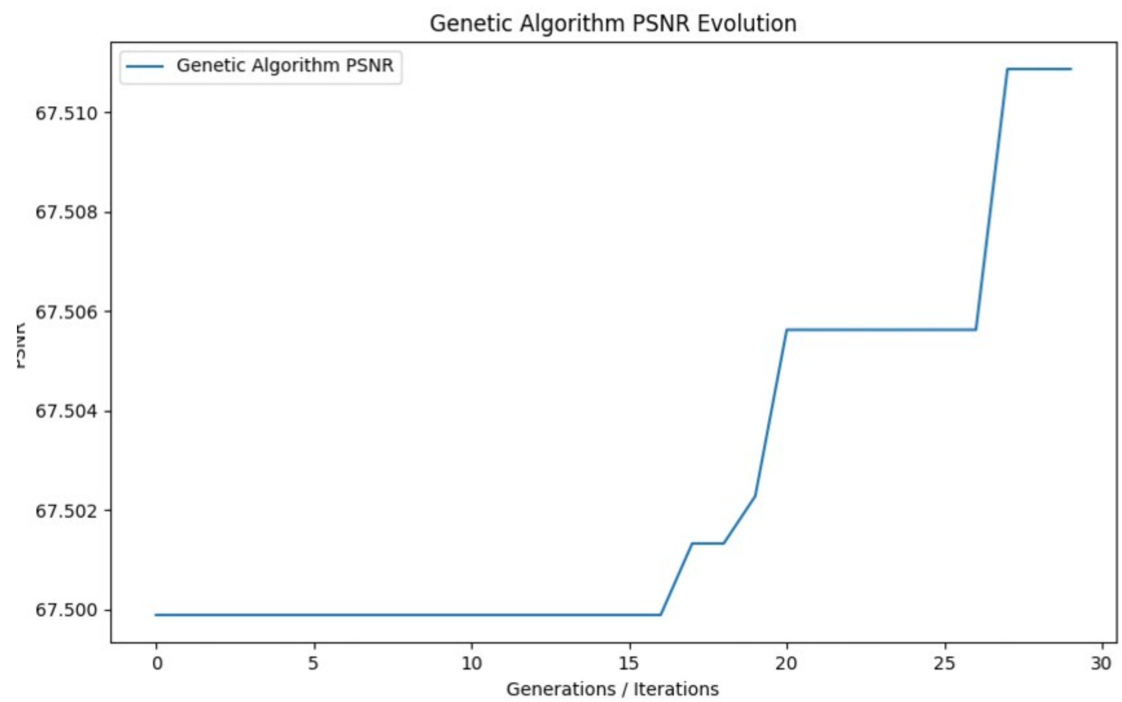
num_particles = 30

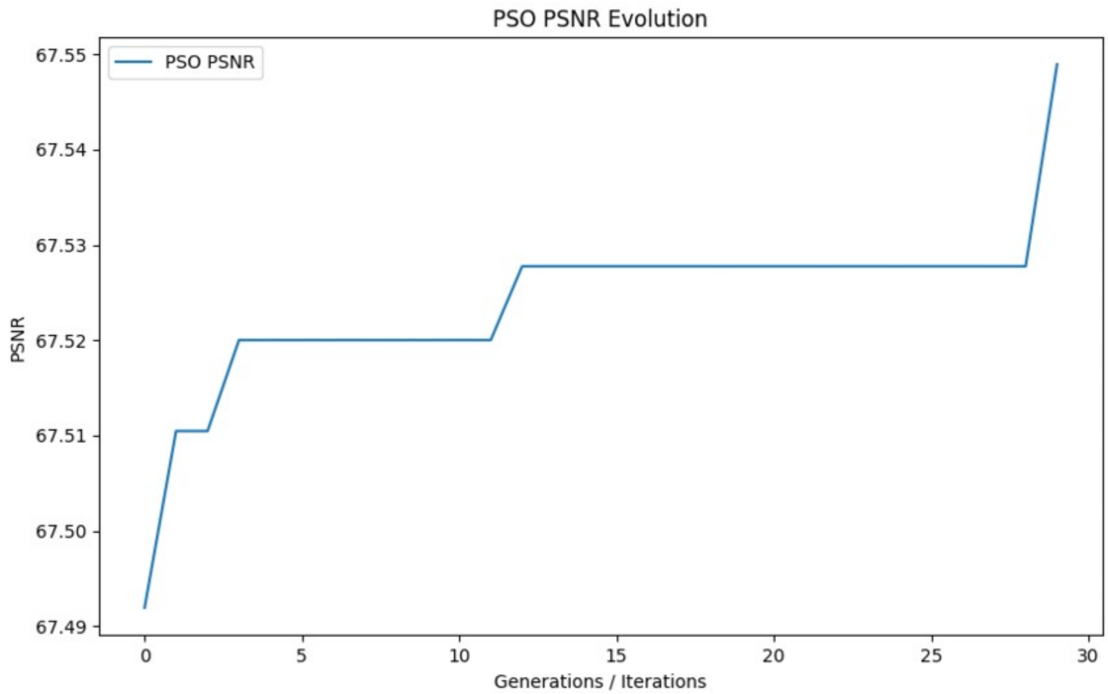
max_iterations = 30

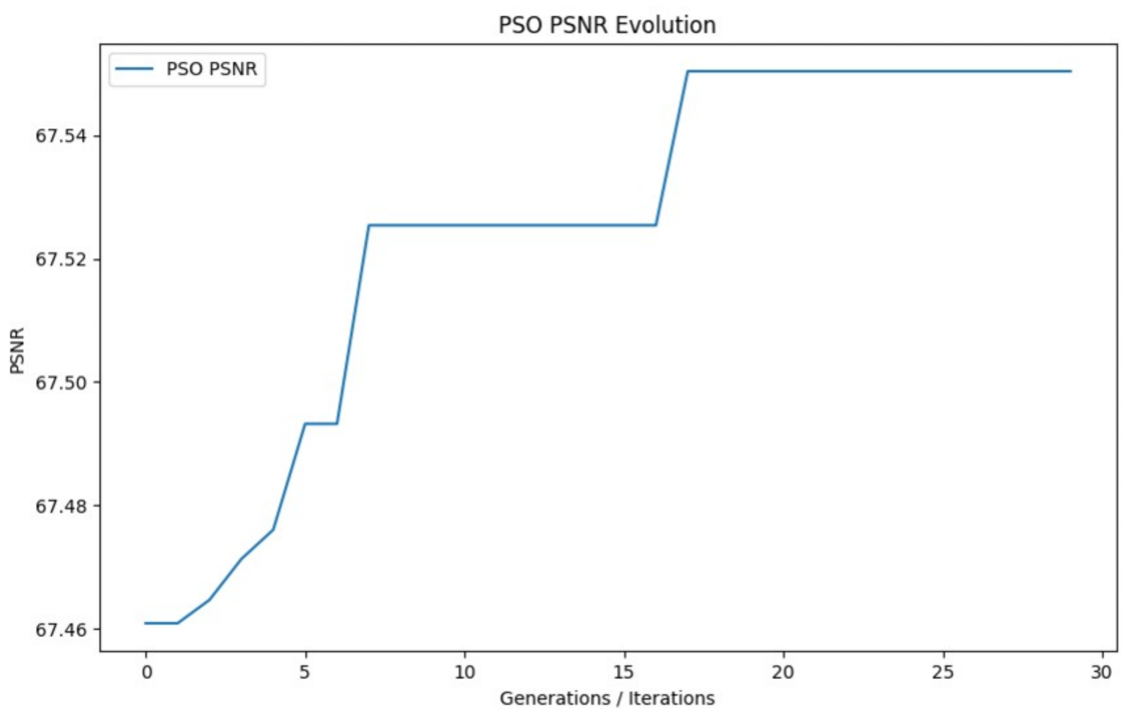
population_size = 30

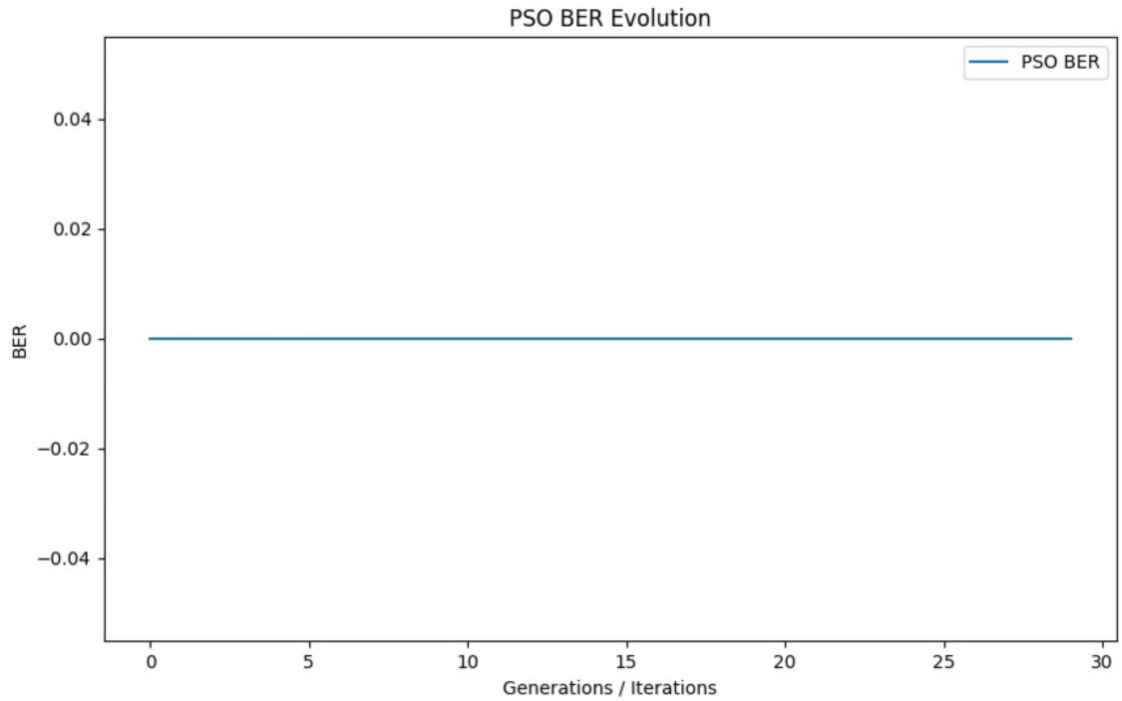
num_generations = 30

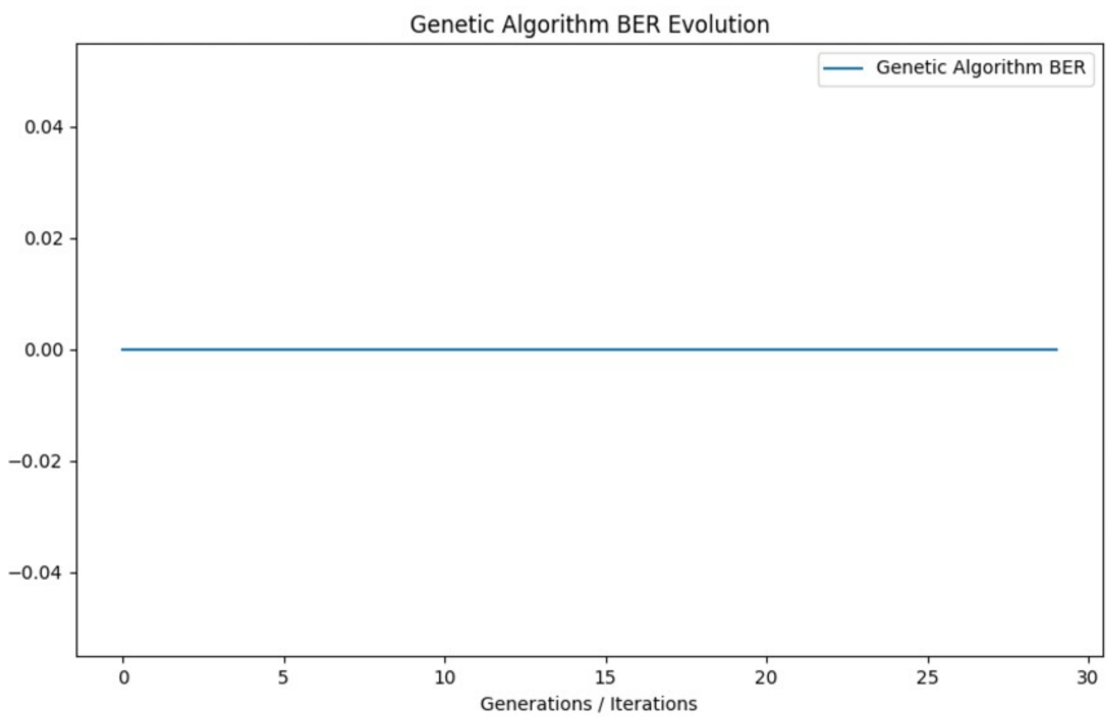
mutation_rate = 0.3

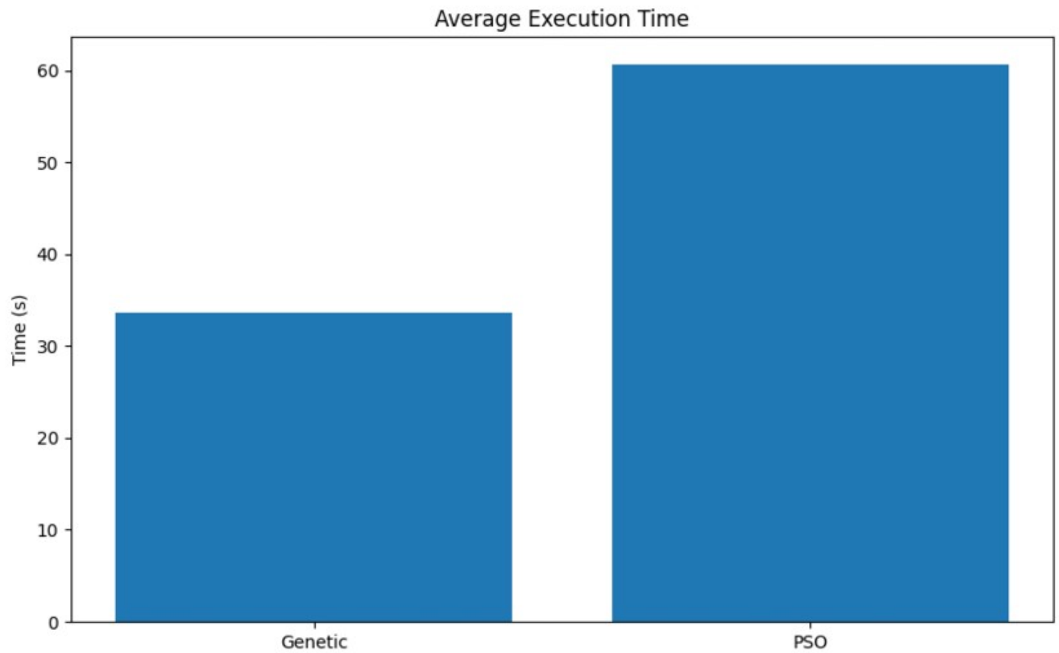



PSO PSNR Evolution
PSO PSNR
67.55
67.54
67.53
67.52
67.51
67.50
67.49
PSNR
0
5
10
15
20
25
30
Generations / Iterations

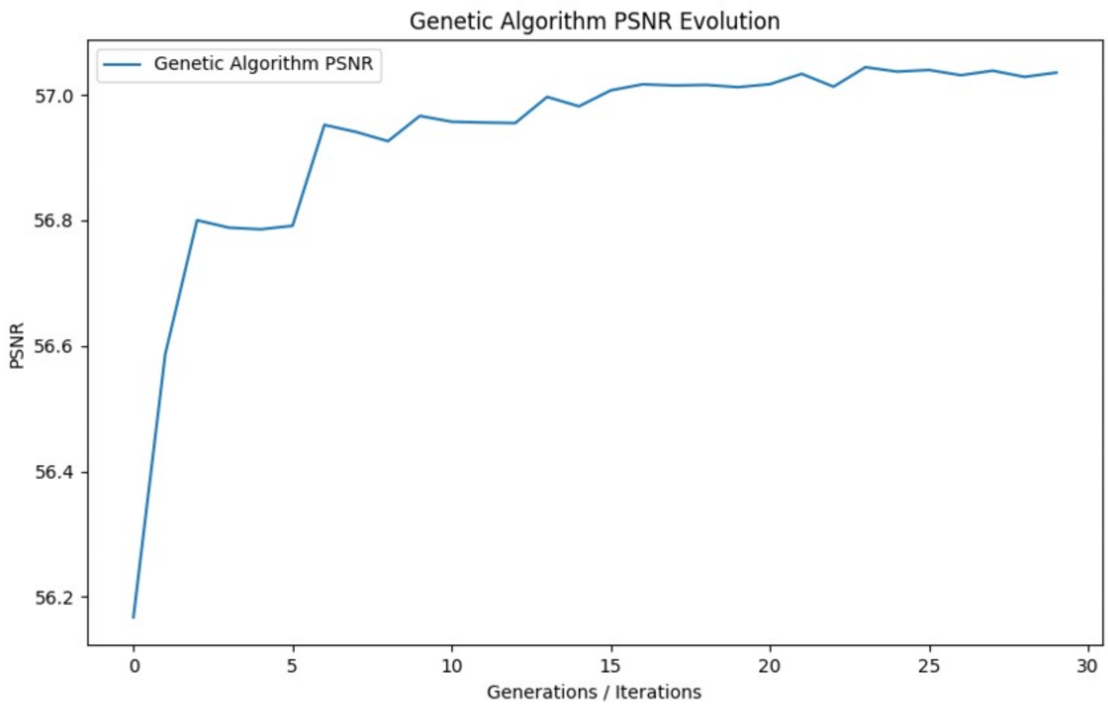

PSO PSNR Evolution
PSO PSNR
67.54
67.52
67.50
67.48
67.46
PSNR
0
5
10
15
20
25
30
Generations / Iterations

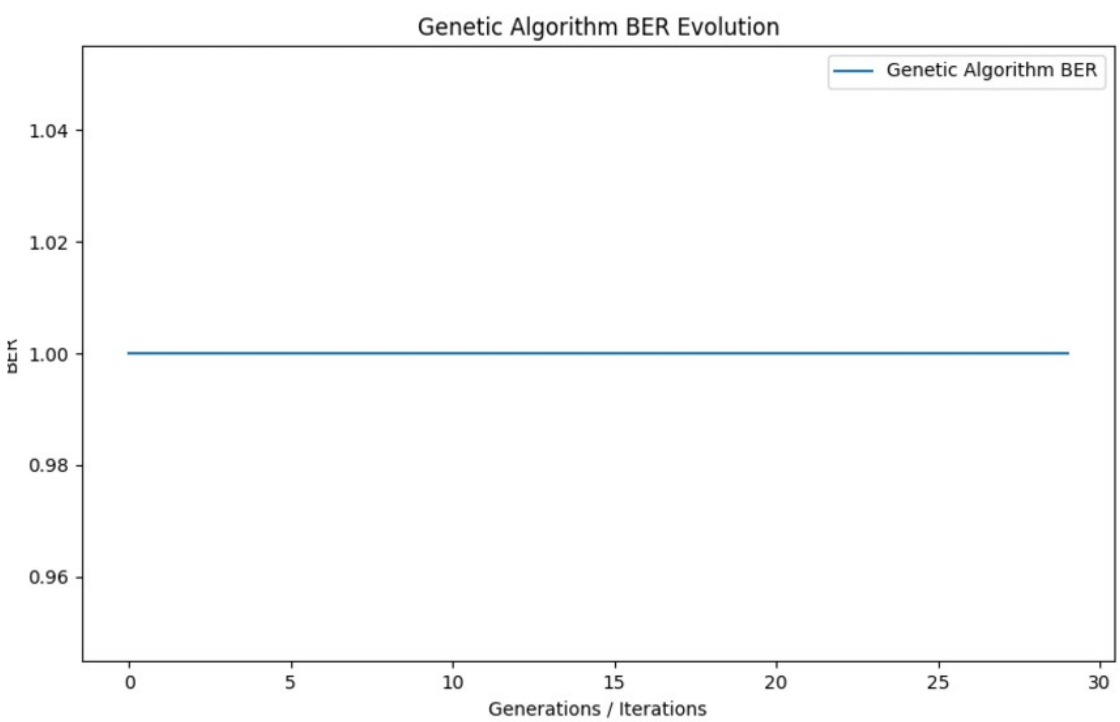

PSO BER Evolution
PSO BER
0.04
0.02
0.00
−0.02
−0.04
BER
0
5
10
15
20
25
30
Generations / Iterations

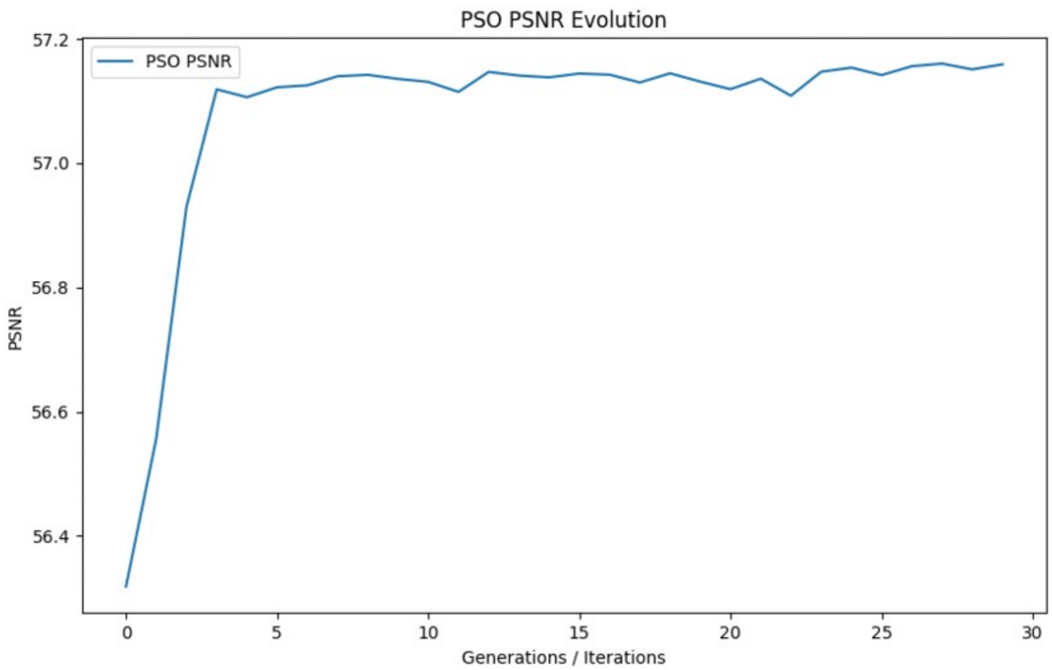

Genetic Algorithm BER Evolution
Genetic Algorithm BER
0.04
0.02
0.00
−0.02
−0.04
0
5
10
15
20
25
30
Generations / Iterations

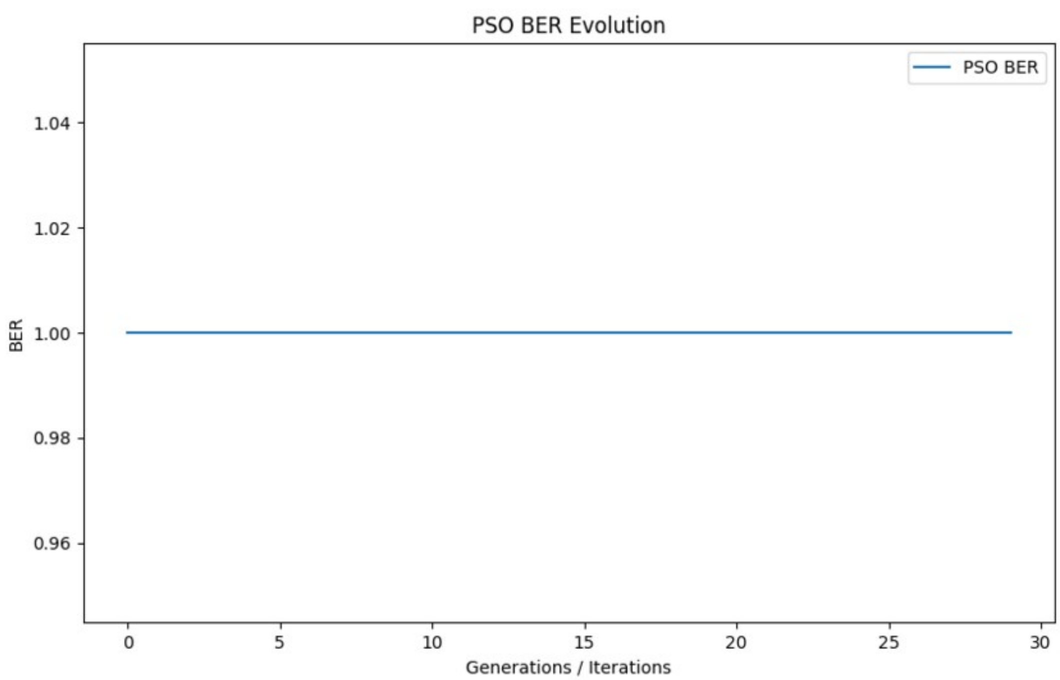


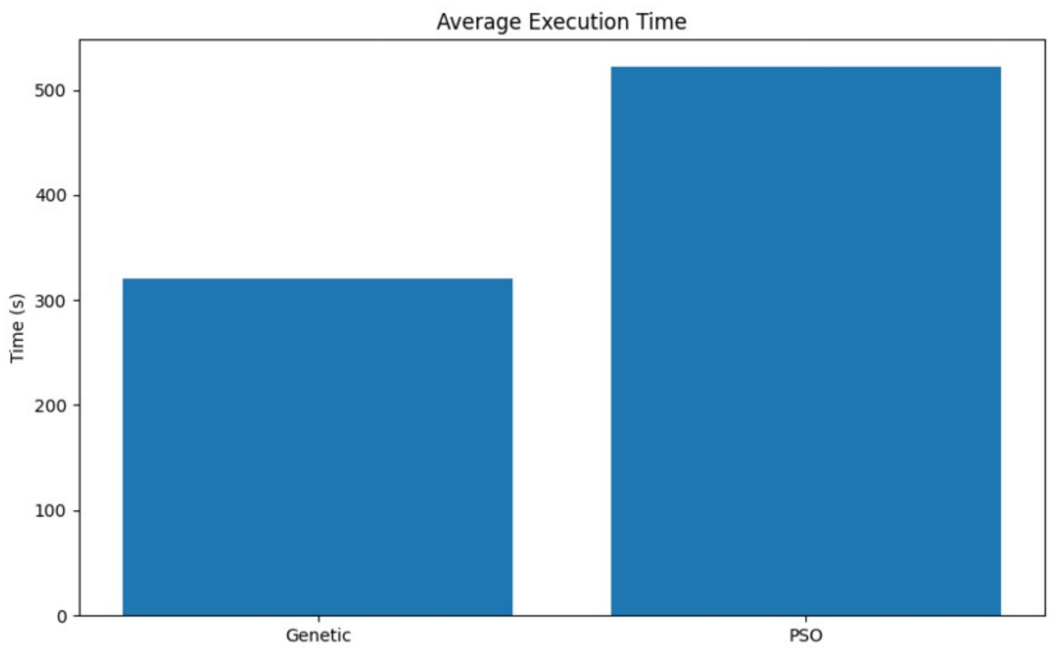
binary_data = ''.join(random.choice(['0', '1']) for _ in range(131072)) 131072 BITS (POLOWA ZDJECIA 512X512)


Genetic Algorithm PSNR Evolution
Genetic Algorithm PSNR
PSNR
Generations / Iterations

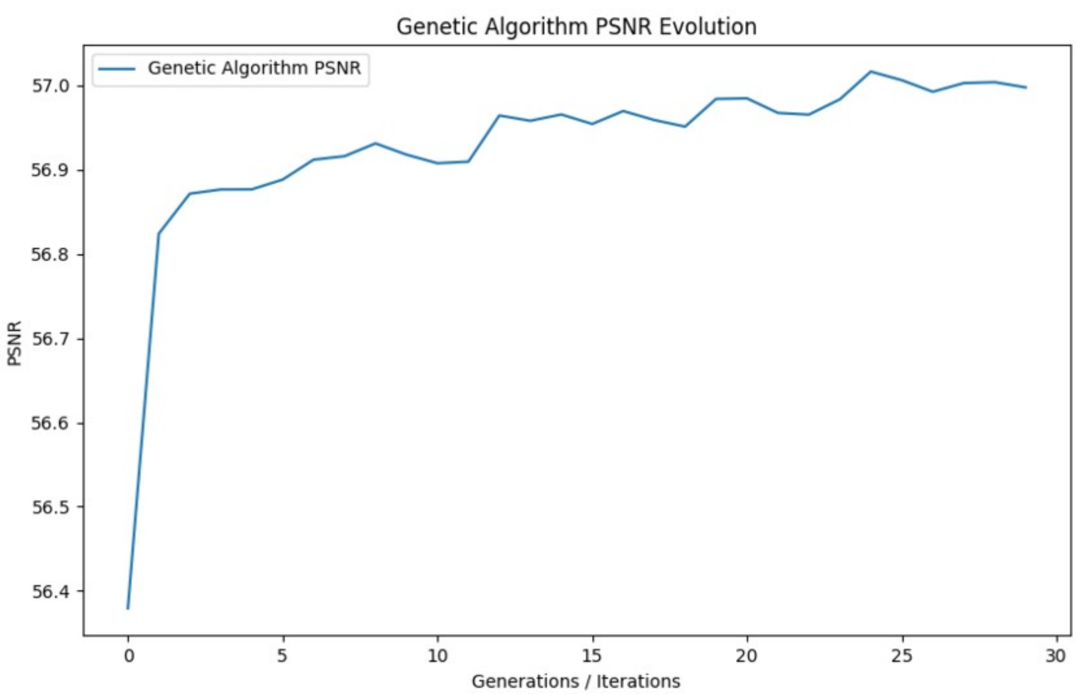

Genetic Algorithm BER Evolution
Genetic Algorithm BER
Generations / Iterations


PSO PSNR Evolution
PSO PSNR
PSNR
57.2
57.0
56.8
56.6
56.4
0
5
10
15
20
25
30
Generations / Iterations

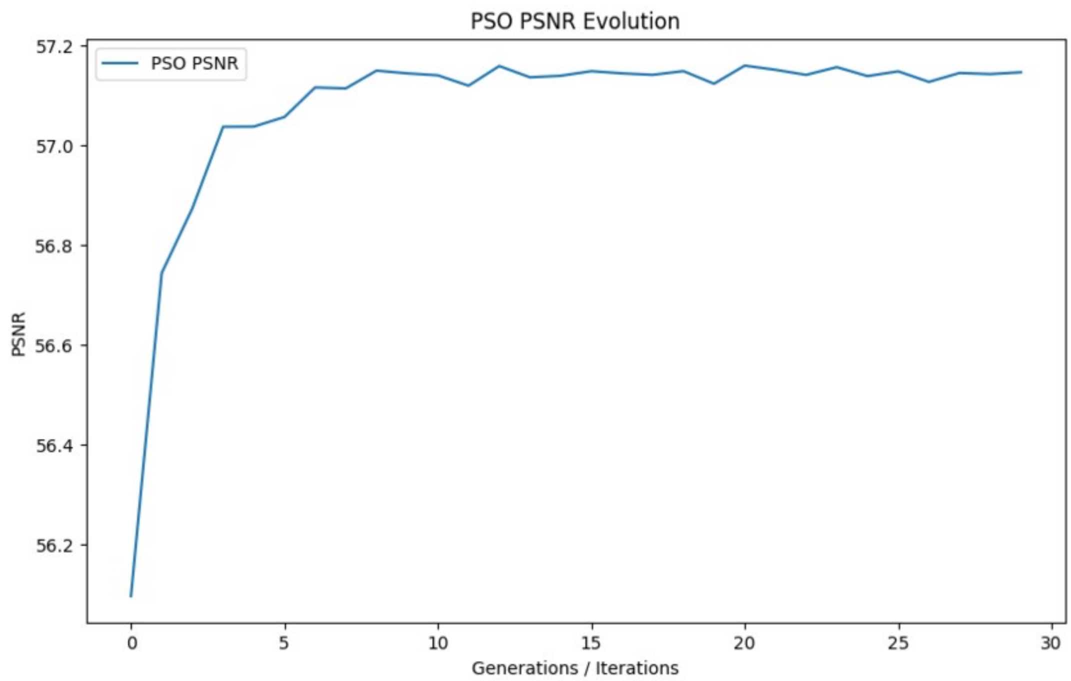

PSO BER Evolution
PSO BER
BER
1.04
1.02
1.00
0.98
0.96
0
5
10
15
20
25
30
Generations / Iterations

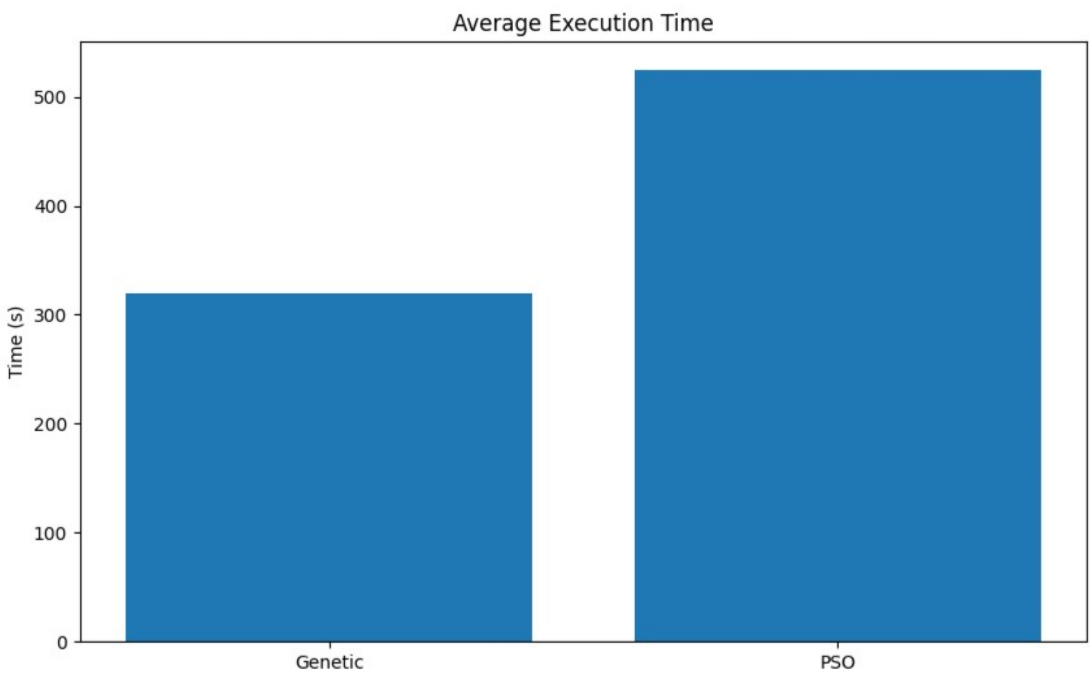


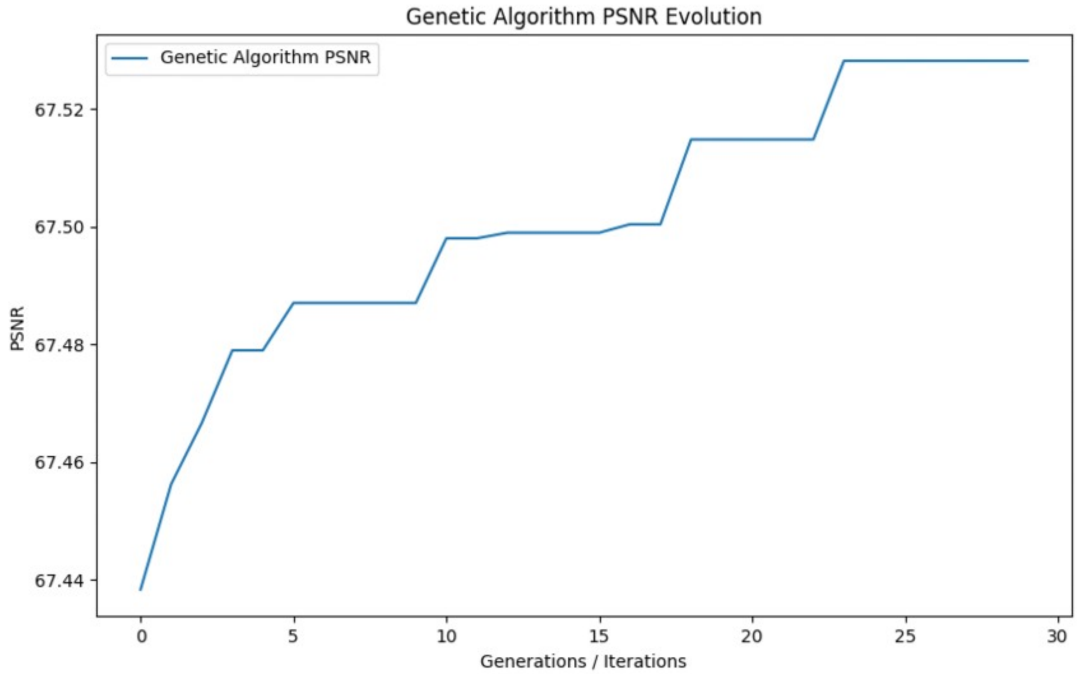
Average Execution Time
Time (s)
0
100
200
300
400
500
Genetic
PSO

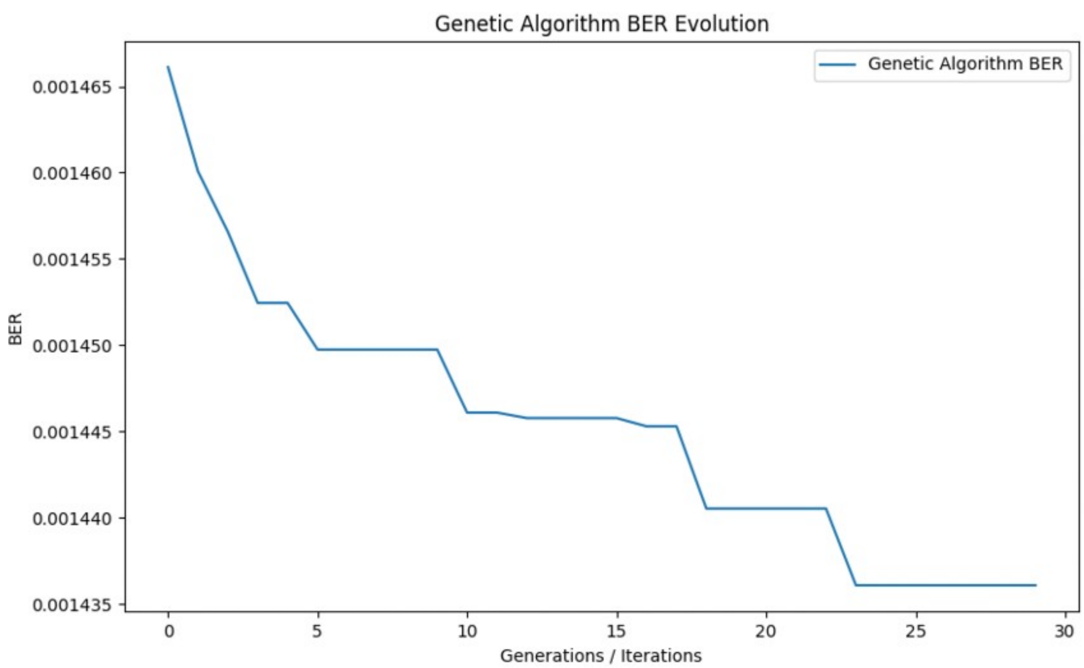


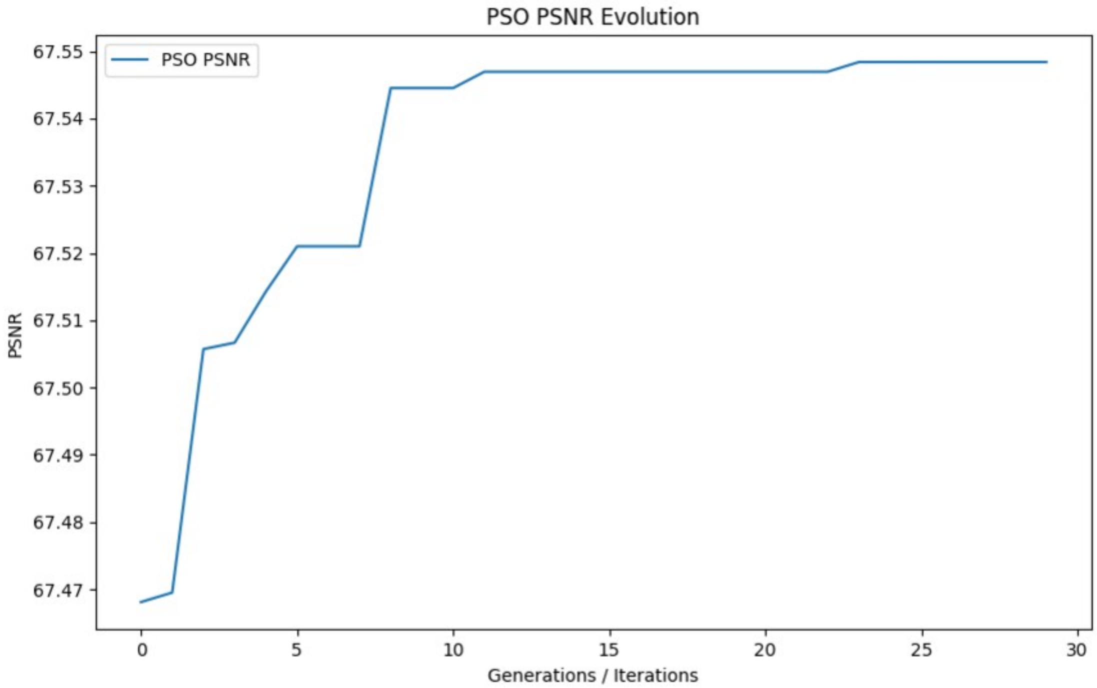
Genetic Algorithm PSNR Evolution
Genetic Algorithm PSNR
PSNR
56.4
56.5
56.6
56.7
56.8
56.9
57.0
0
5
10
15
20
25
30
Generations / Iterations

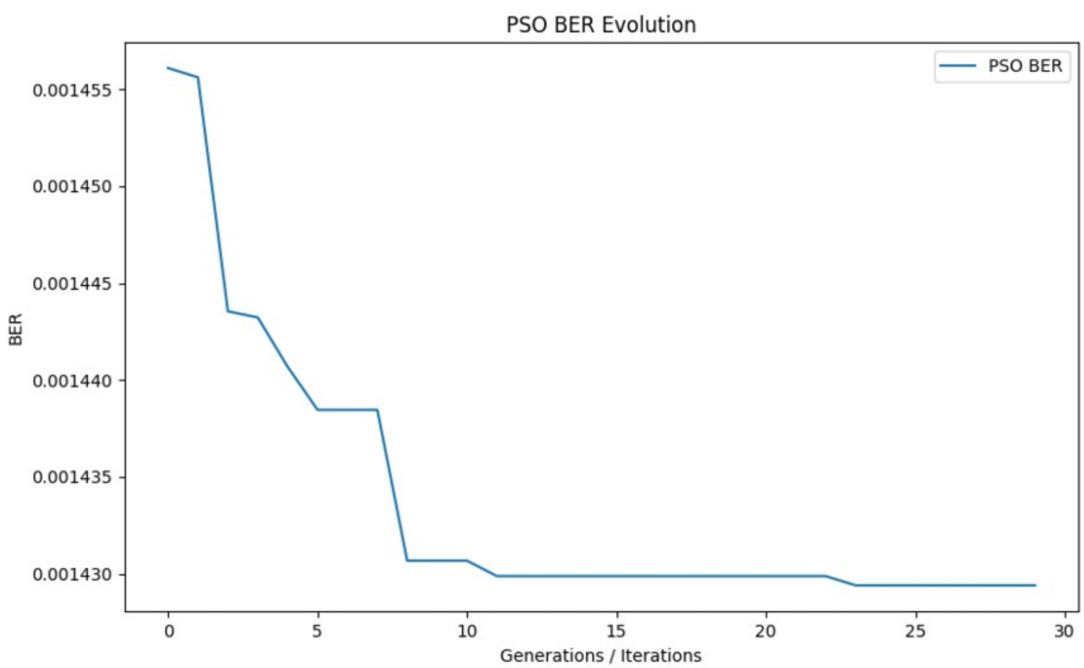

PSO PSNR Evolution
PSO PSNR
PSNR
Generations / Iterations
57.2
57.0
56.8
56.6
56.4
56.2
0
5
10
15
20
25
30

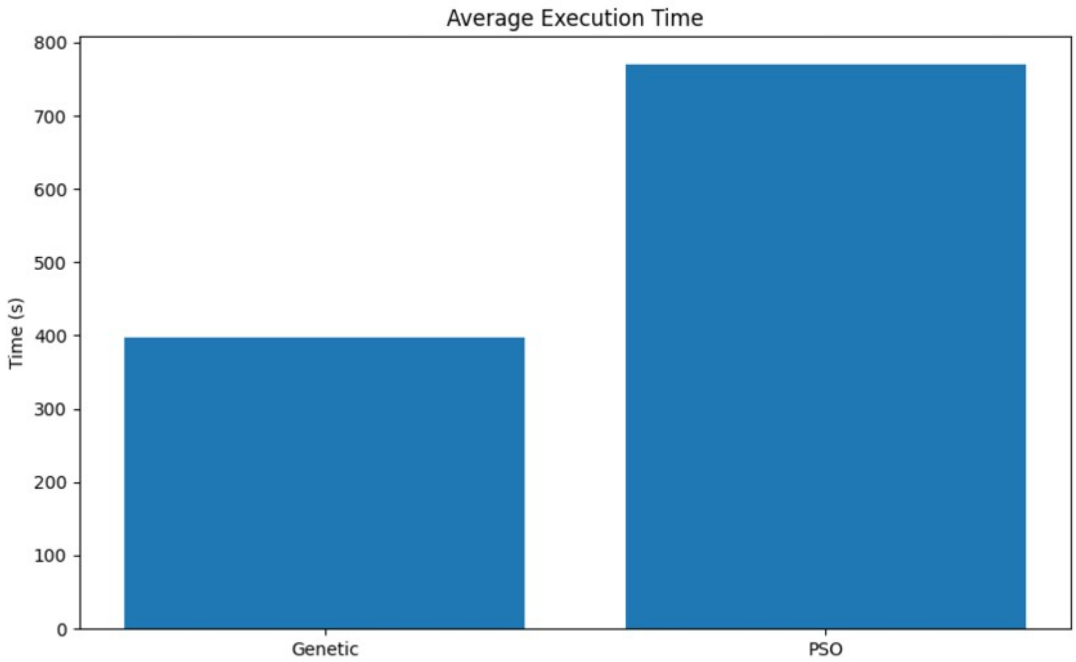

Average Execution Time
Time (s)
500
400
300
200
100
0
Genetic
PSO


Genetic Algorithm PSNR Evolution
Genetic Algorithm PSNR
67.52
67.50
67.48
67.46
67.44
PSNR
0
5
10
15
20
25
30
Generations / Iterations

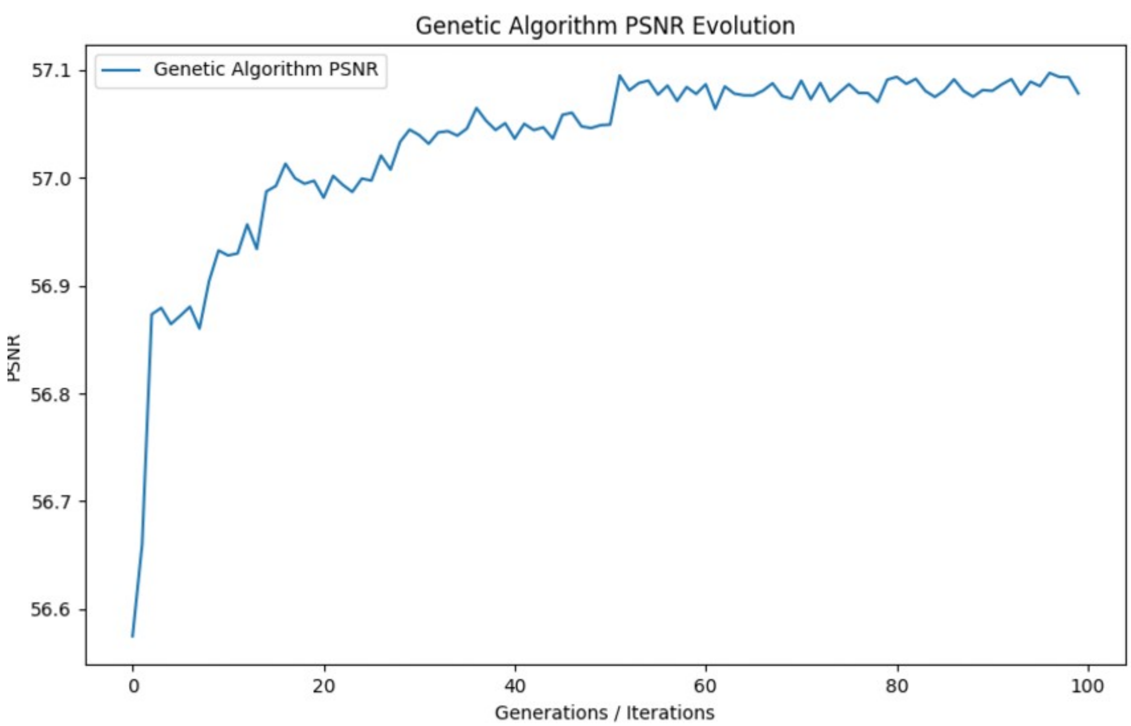

Genetic Algorithm BER Evolution
Genetic Algorithm BER
0.001465
0.001460
0.001455
0.001450
0.001445
0.001440
0.001435
BER
0
5
10
15
20
25
30
Generations / Iterations

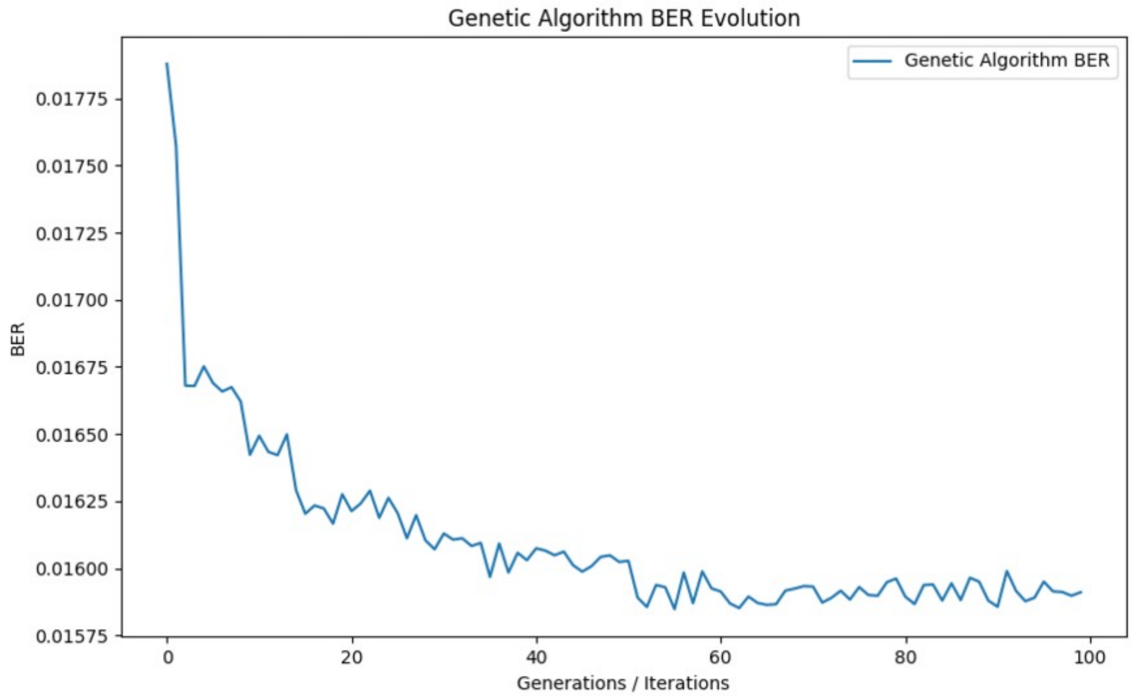

PSO PSNR Evolution
PSO PSNR
PSNR
67.55
67.54
67.53
67.52
67.51
67.50
67.49
67.48
67.47
0
5
10
15
20
25
30
Generations / Iterations

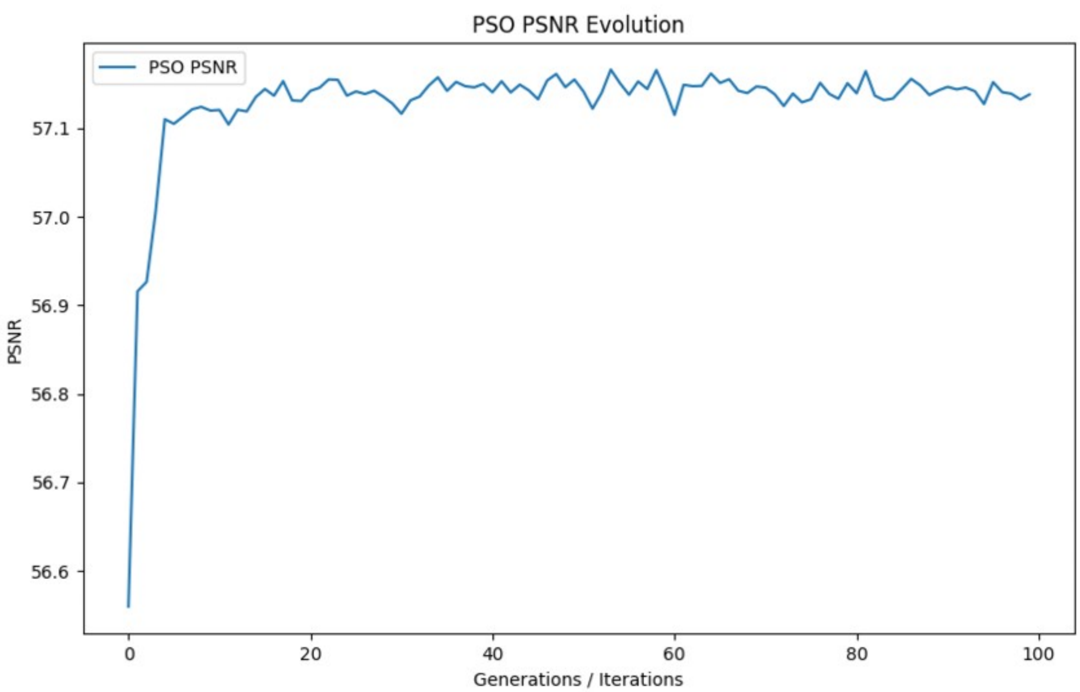

PSO BER Evolution
PSO BER
BER
0.001455
0.001450
0.001445
0.001440
0.001435
0.001430
0
5
10
15
20
25
30
Generations / Iterations

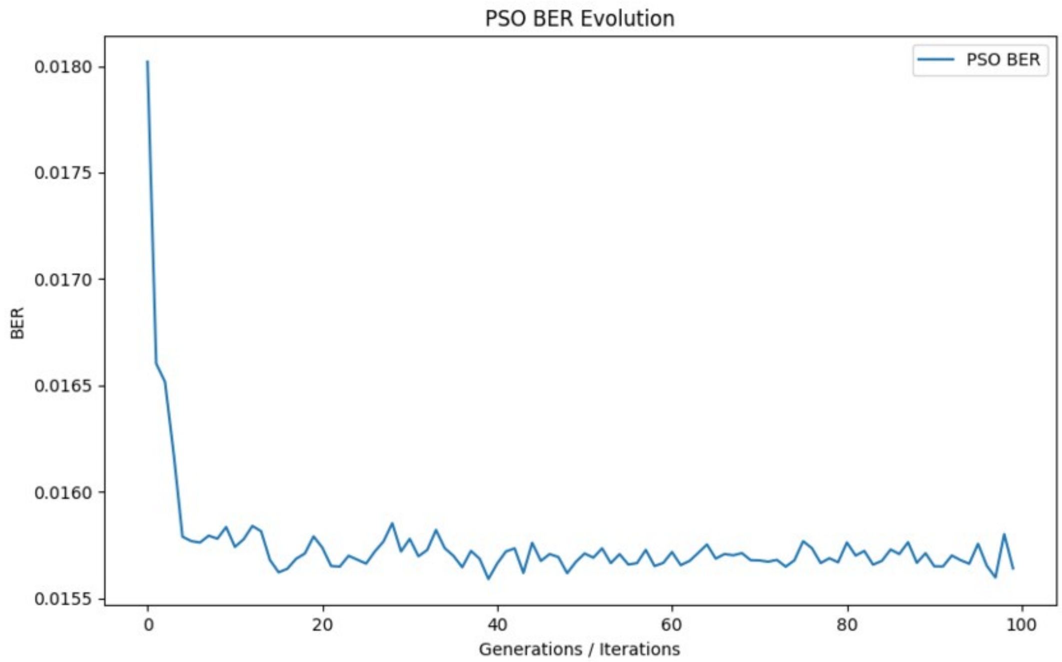


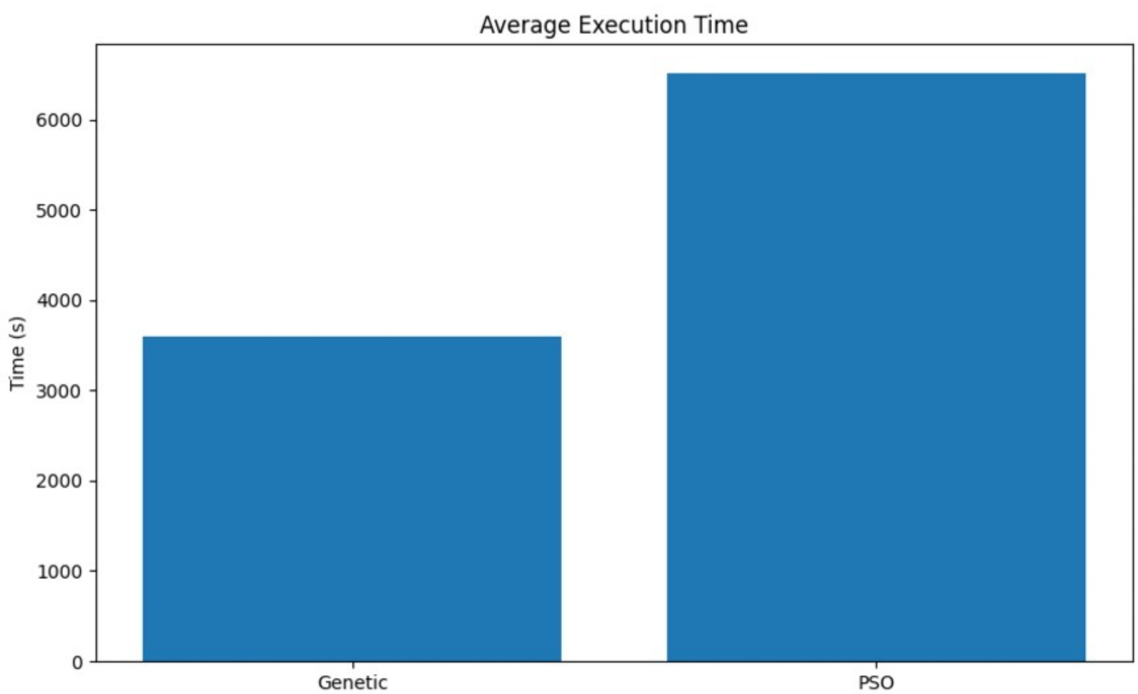
ZWIEKSZONO POPULACJE DO 50 oraz epoki do 100



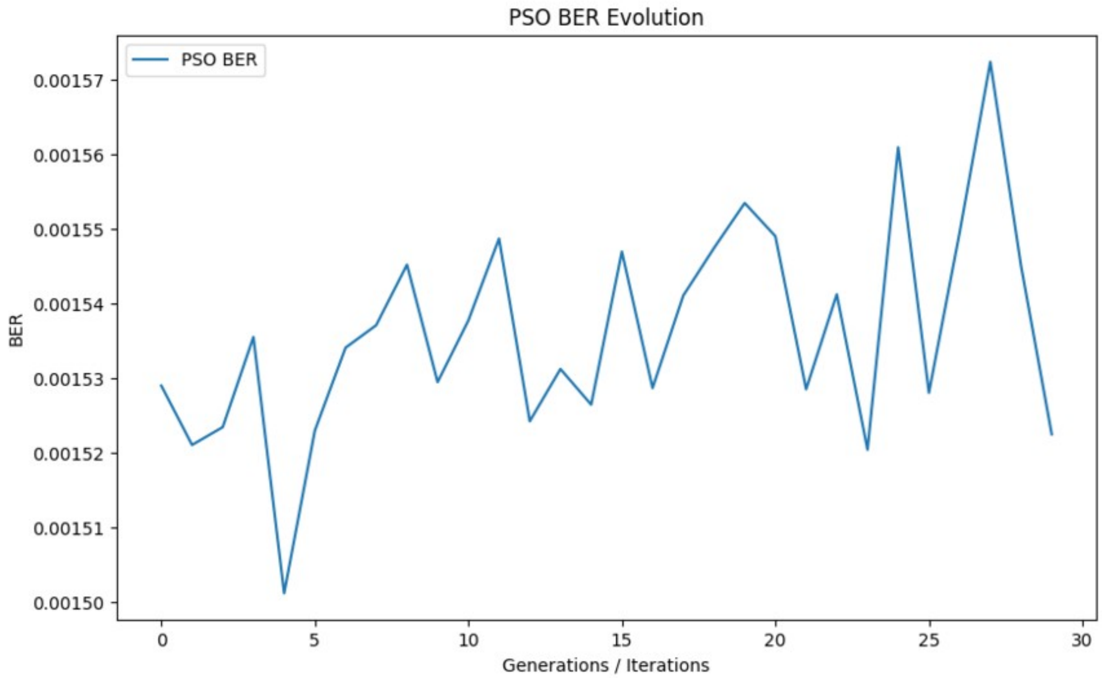

Genetic Algorithm BER Evolution
Genetic Algorithm BER
0.01775
0.01750
0.01725
0.01700
0.01675
0.01650
0.01625
0.01600
0.01575
BER
0
20
40
60
80
100
Generations / Iterations


PSO PSNR Evolution
PSO PSNR
57.1
57.0
56.9
56.8
56.7
56.6
PSNR
0
20
40
60
80
100
Generations / Iterations

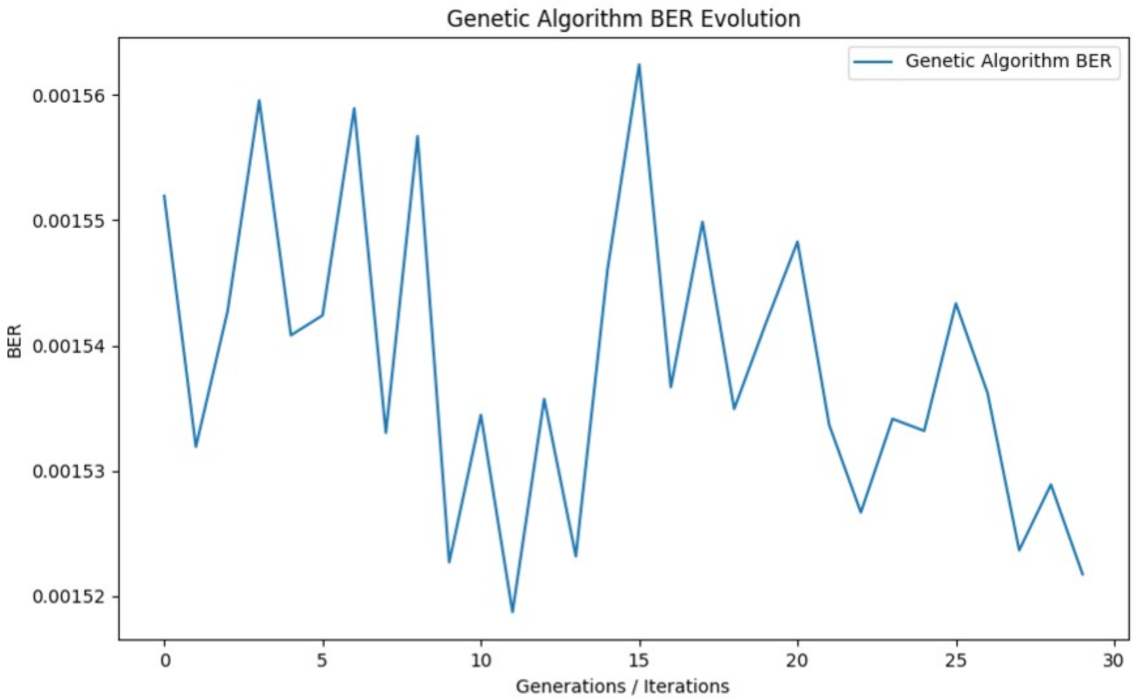


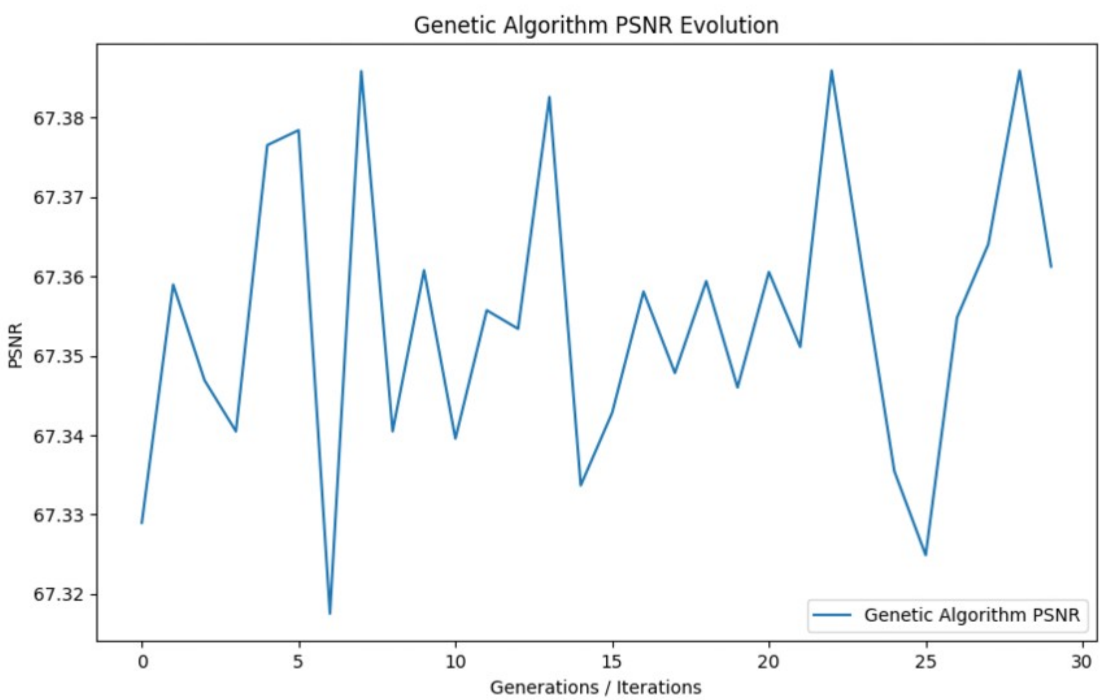


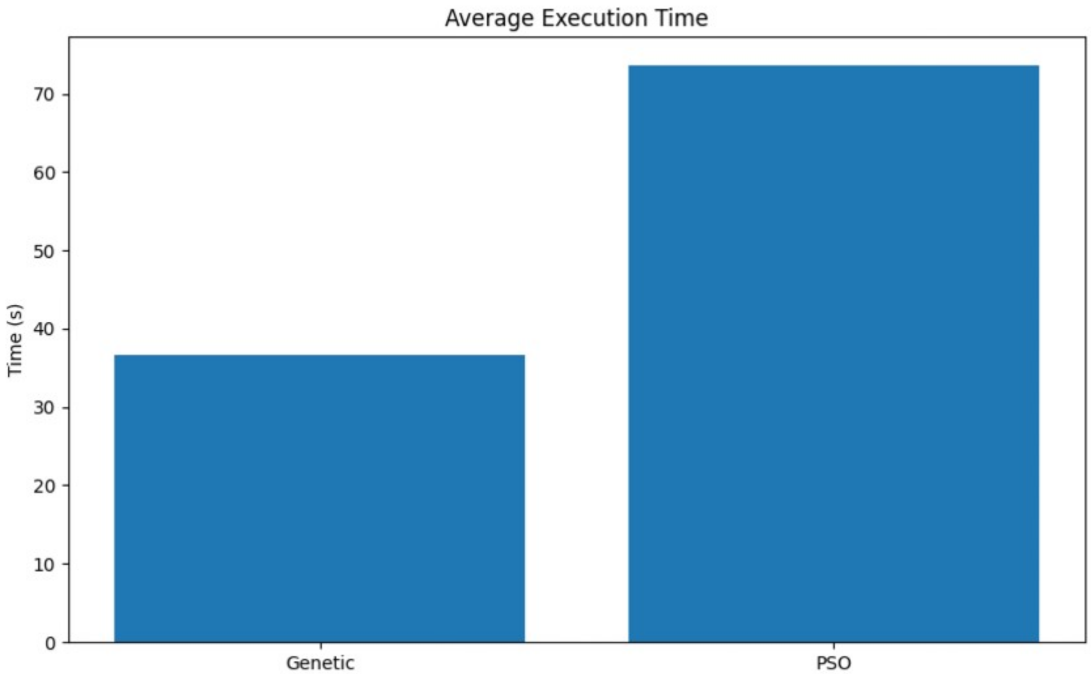
Po optymalizacji 30 epok, 30 populacja, 6400 bitow


PSO BER Evolution
PSO BER
BER
Generations / Iterations


PSO PSNR Evolution
PSO PSNR
PSNR
Generations / Iterations

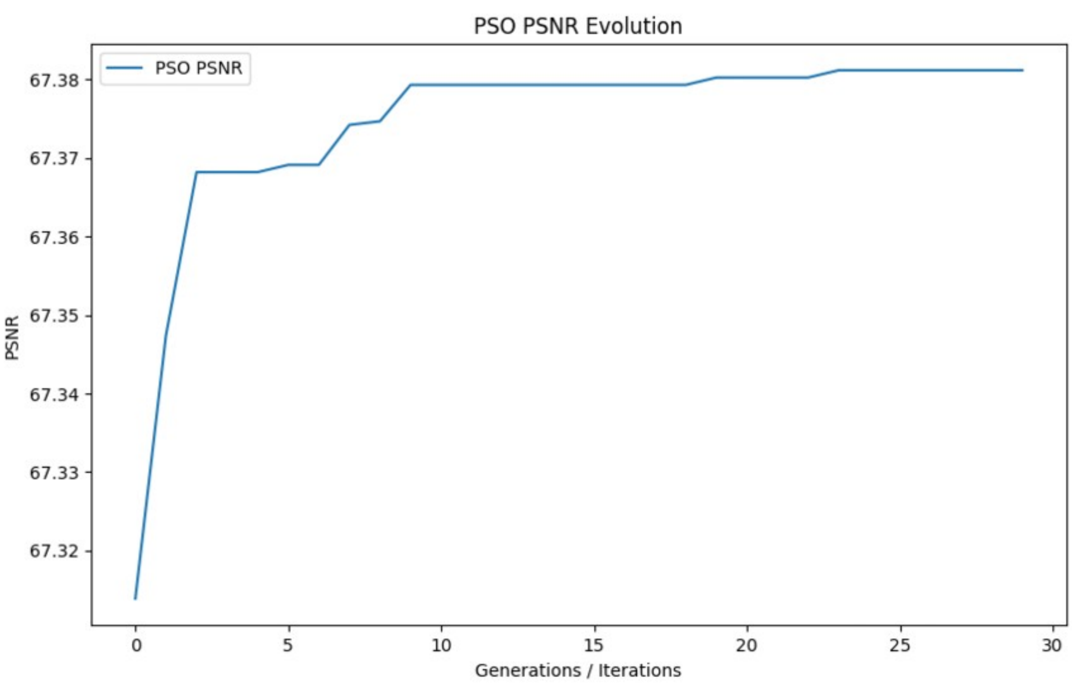

Genetic Algorithm BER Evolution
Genetic Algorithm BER
BER
0.00156
0.00155
0.00154
0.00153
0.00152
0
5
10
15
20
25
30
Generations / Iterations

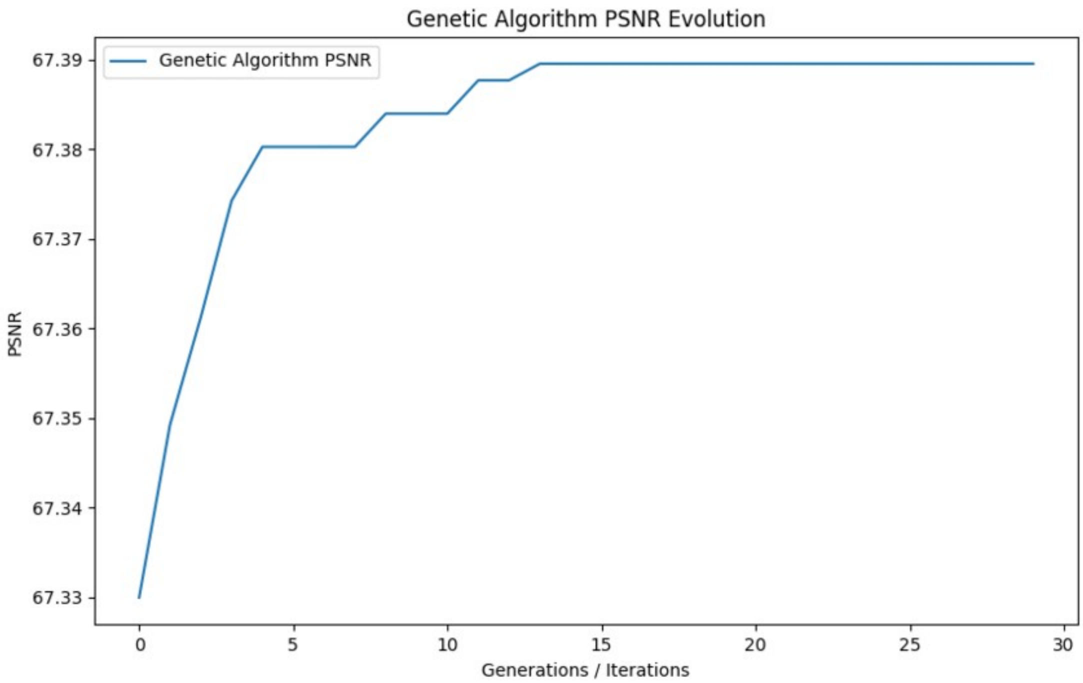

Genetic Algorithm PSNR Evolution
PSNR
67.38
67.37
67.36
67.35
67.34
67.33
67.32
Genetic Algorithm PSNR
0
5
10
15
20
25
30
Generations / Iterations

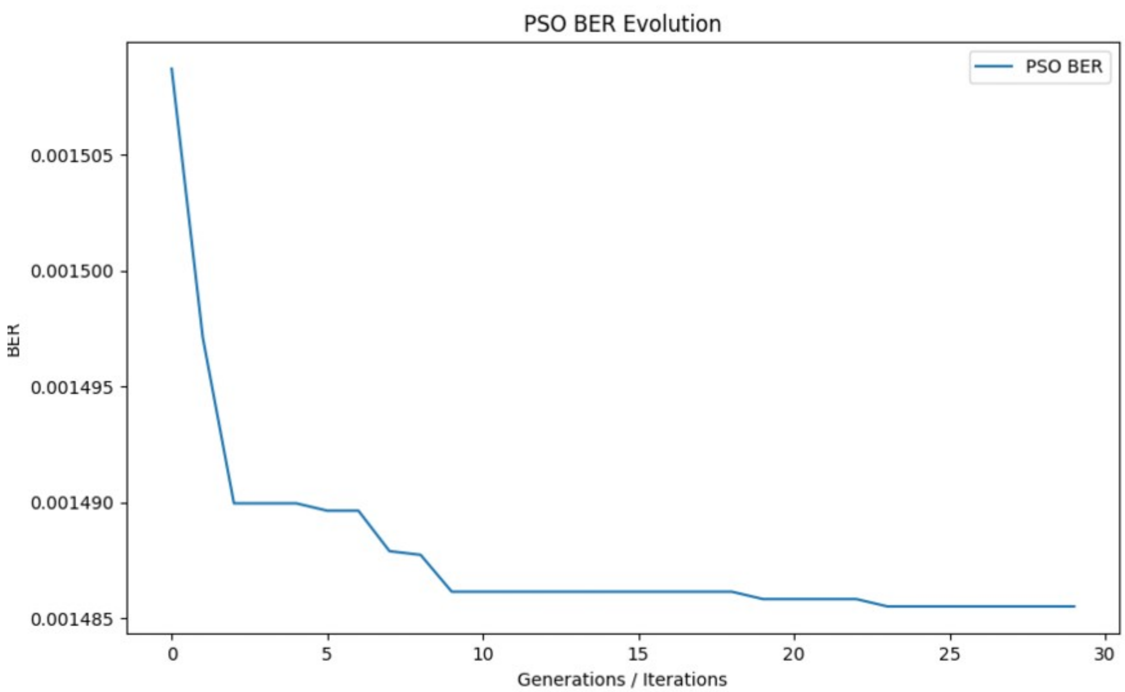


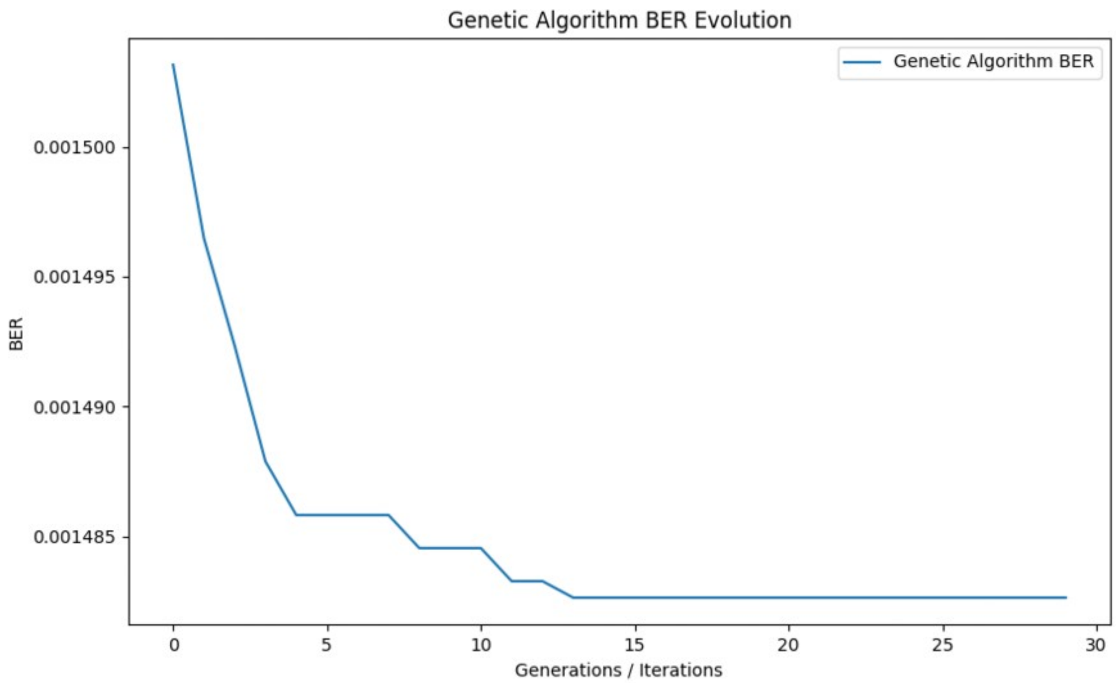
Dane do ukrycia byly losowo generowane dla kazdej generacji!!!!

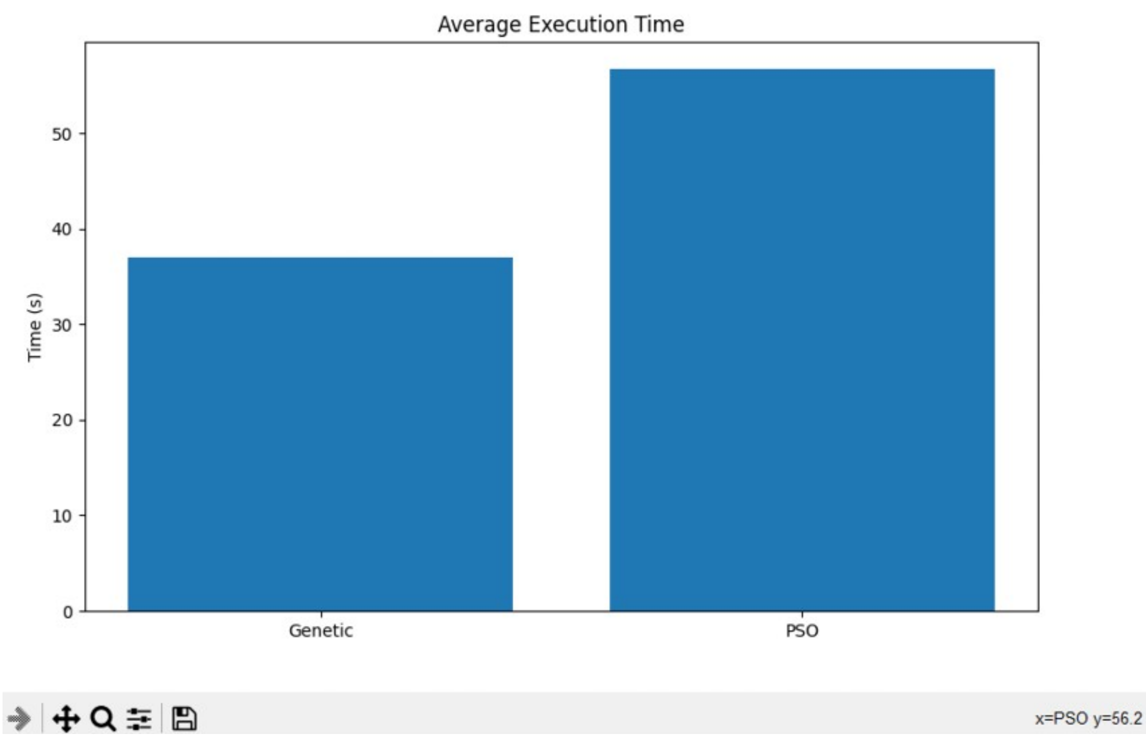
W PSO BRAKOWALO .copy() przy przypisywaniu najlepszej czastki




Genetic Algorithm PSNR Evolution
Genetic Algorithm PSNR
PSNR
67.39
67.38
67.37
67.36
67.35
67.34
67.33
0
5
10
15
20
25
30
Generations / Iterations


PSO BER Evolution
PSO BER
BER
0.001505
0.001500
0.001495
0.001490
0.001485
0
5
10
15
20
25
30
Generations / Iterations





Ok. 5 sekund poprawy sredniego czasu po przepisaniu ze stringow na array[0,1]

TESTY

populacja : 30

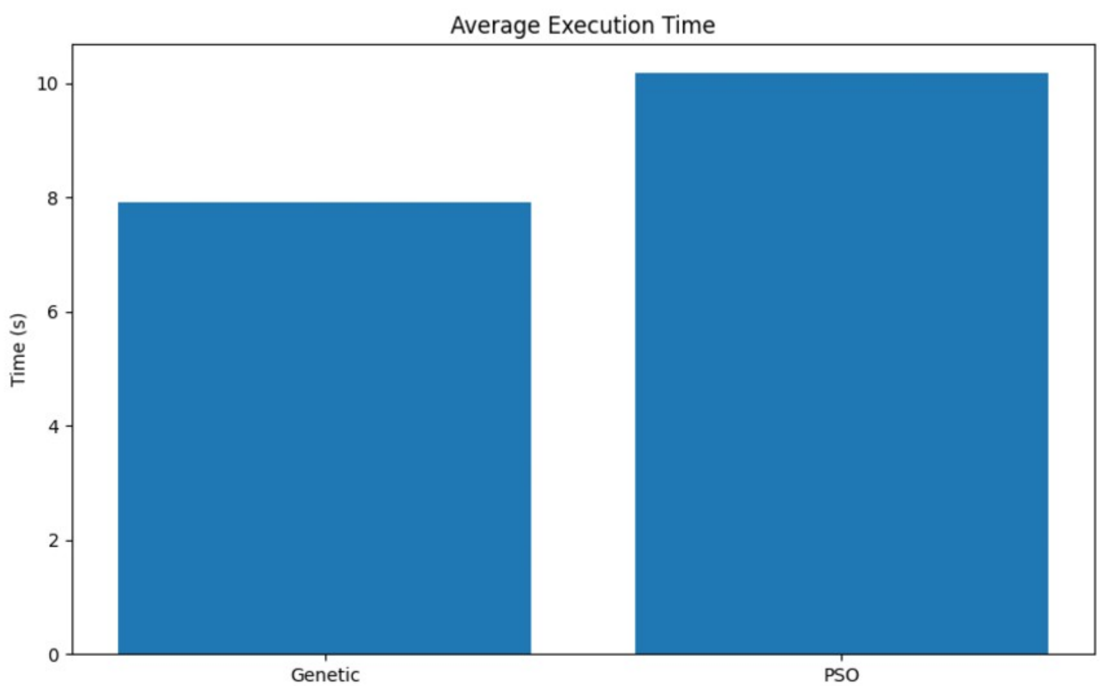
generacje : 30

mutation_rate = 0.3

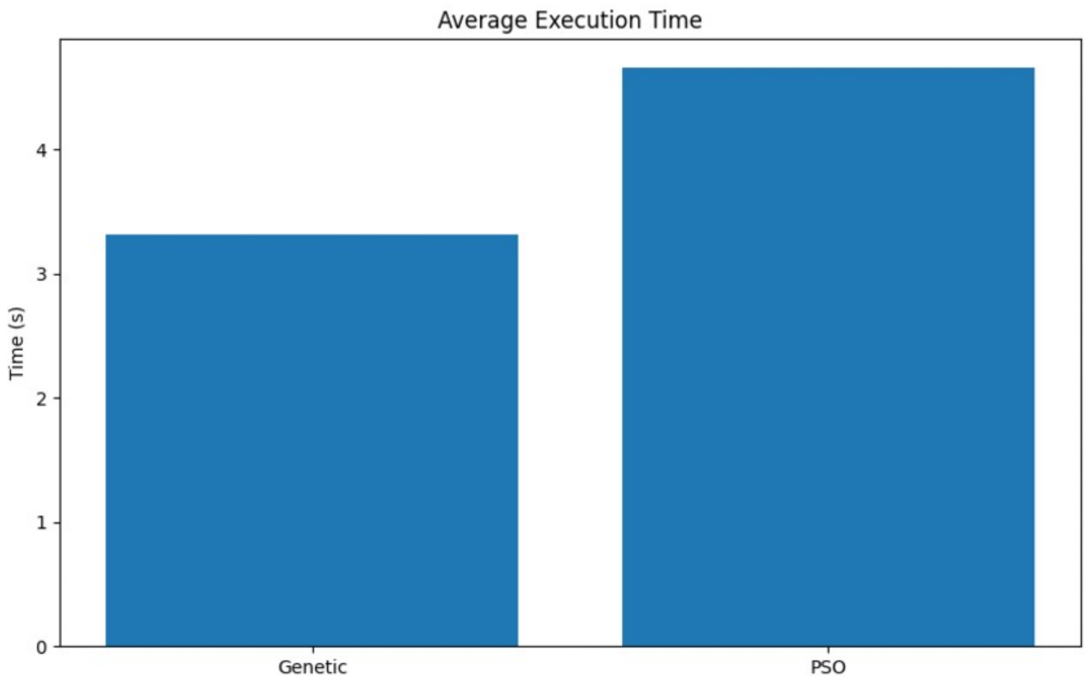
zdjecie : 256x256 = 65,536 bits

ukryte = 20000 bits

czasy : PSO 10s GENETIC 8s



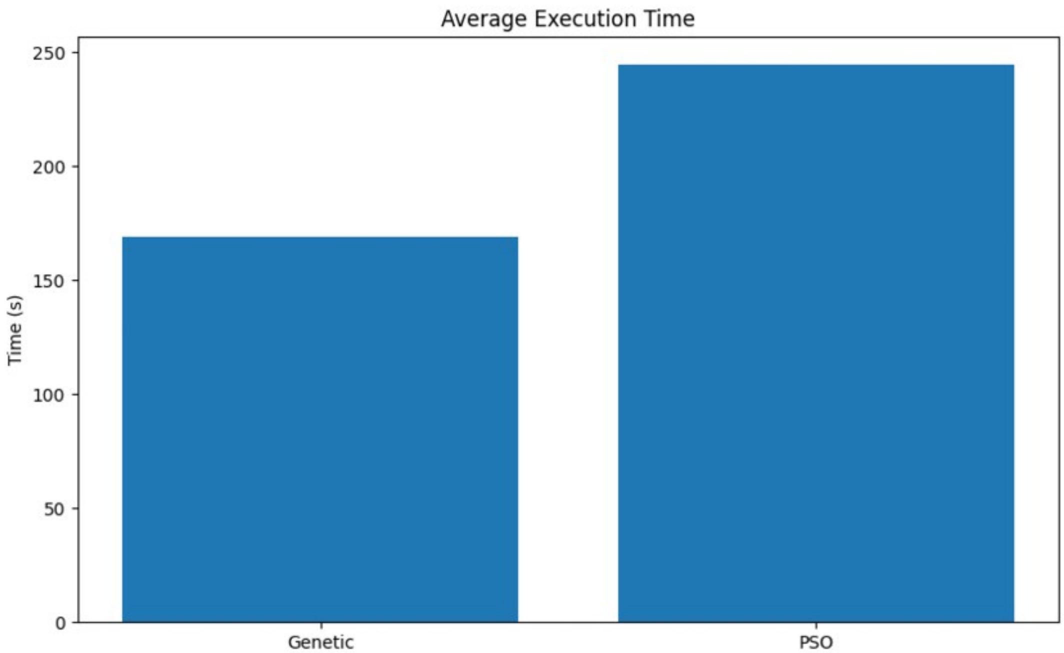
Po zakomentowaniu czesci od Dekodowania (wykorzystywana do teraz jako sprawdzenie poprawnosci odczytanych danych)



PSO : 4.64 GENETYK : 3.25

TE SAME DANE WIEKSZY OBRAZ 512x512 = 262,144 bits

UKRYTE : 1/3 * 262,144 bits = 87400 bits



PSO : 243.4 GENETIC : 169.5

populacja : 50

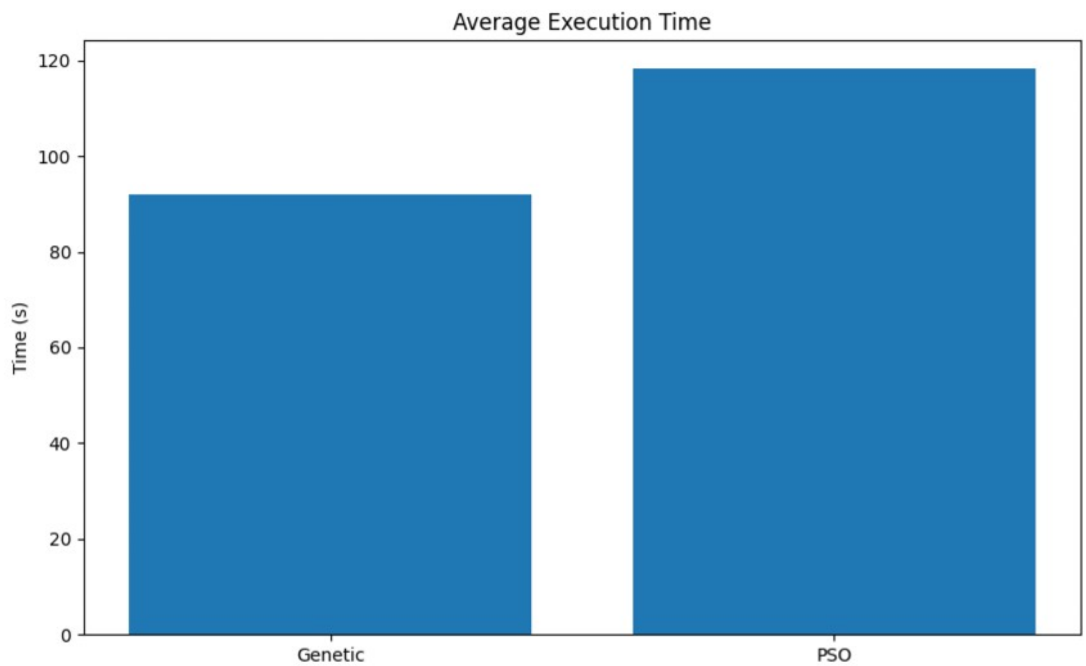
generacje : 100

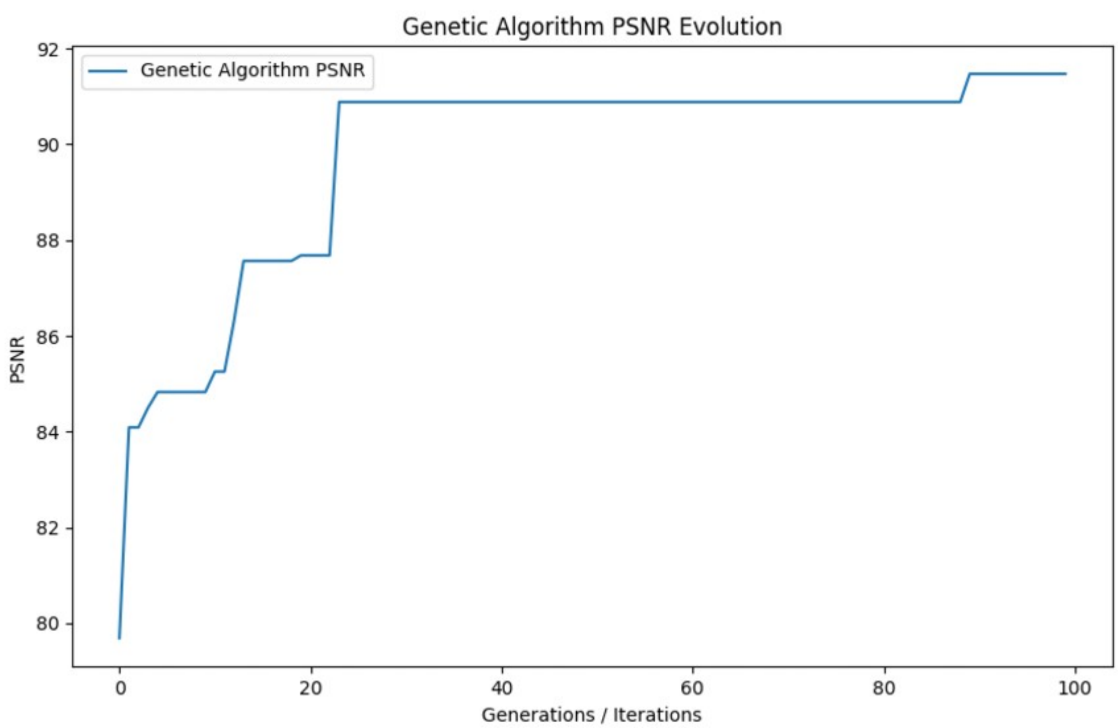
mutation_rate = 0.3

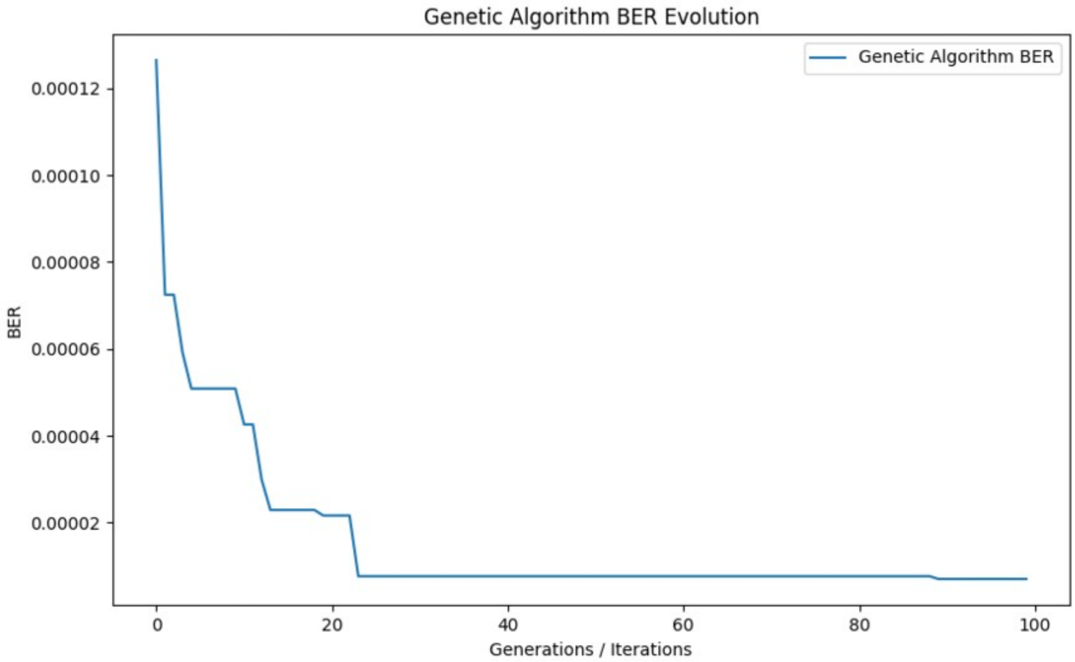
zdjecie : 256x256 = 65,536 bits

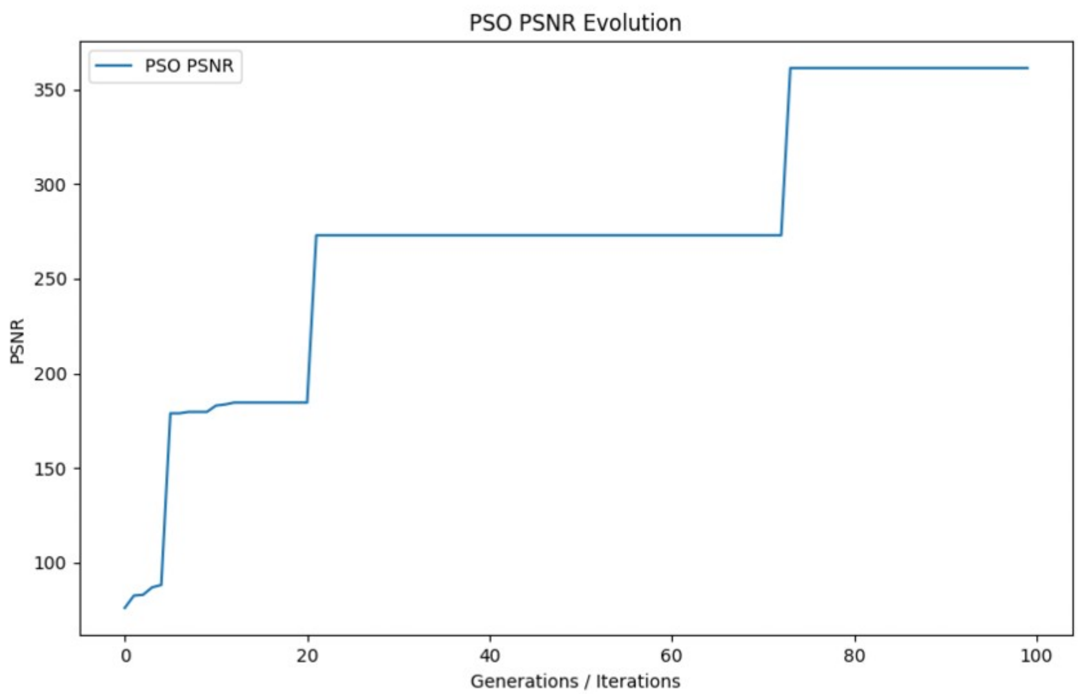
ukryte = 20000 bits

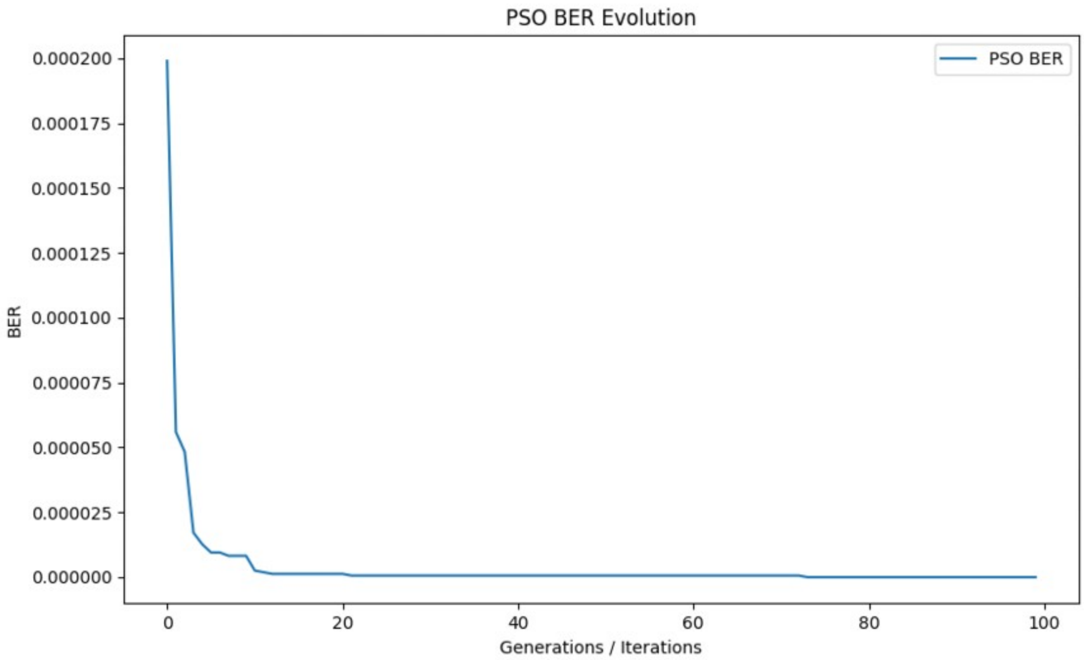
CZASY : PSO : 119.5 GENETIC : 91.5


Average Execution Time
Time (s)
0
20
40
60
80
100
120
Genetic
PSO


Genetic Algorithm PSNR Evolution
Genetic Algorithm PSNR
PSNR
80
82
84
86
88
90
92
0
20
40
60
80
100
Generations / Iterations


Genetic Algorithm BER Evolution
Genetic Algorithm BER
0.00012
0.00010
0.00008
0.00006
0.00004
0.00002
BER
0
20
40
60
80
100
Generations / Iterations


PSO PSNR Evolution
PSO PSNR
350
300
250
200
150
100
PSNR
0
20
40
60
80
100
Generations / Iterations



zdjecie : 512x512 = 262,144 bits

ukryte = 87500 bits

CZASY : PSO : 119.5 GENETIC : 91.5

PO ZROWNOLEGLENIU:

* DODANO Pooling

```
def parallel_fitness(population, image, binary_data, delimiter):
    with Pool() as pool:
        fitness_scores = pool.map(fitness_function_array,
                                  [(chromosome, image, binary_data, delimiter) for chromosome in population])
    return fitness_scores
```

* zmieniono z Threading.Thread na multiprocessing.Process

* przekazano wiele atrybutow jako parametr, w celu zniwelowania ponownych obliczen

Epoki : 30

Populacje : 30

zdjecie : 512x512 = 262,144 bits

ukryte = 87500 bits

Genetic Time: 63.429412841796875 seconds

PSO Time: 63.681725025177 seconds

Epoki : 100

Populacje : 50

Genetic Time: 258.55763602256775 seconds

PSO Time: 259.01299810409546 seconds

Testy porownawcze na 4 procesach dla populacji = 50 oraz epok = 100 wykazaly, ze rownoleglenie nie ma sensu dla malych populacji – czas sekwencyjny byl nizszy niz zrownoleglony:

```
Starting comparison of algorithms across different image sizes...

Running for matrix size: 120
Starting run for matrix size 120 with 4 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 0.2876 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 0.7161 seconds.
  - Starting parallel genetic algorithm with 4 processes...
  - Parallel genetic algorithm completed in 35.9950 seconds.
  - Starting parallel PSO algorithm with 4 processes...
  - Parallel PSO algorithm completed in 36.4752 seconds.
Running for matrix size: 240
Starting run for matrix size 240 with 4 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 2.1905 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 3.8687 seconds.
  - Starting parallel genetic algorithm with 4 processes...
  - Parallel genetic algorithm completed in 37.8793 seconds.
  - Starting parallel PSO algorithm with 4 processes...
  - Parallel PSO algorithm completed in 39.1787 seconds.
Running for matrix size: 360
```

```
Starting run for matrix size 360 with 4 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 5.7399 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 11.7645 seconds.
  - Starting parallel genetic algorithm with 4 processes...
  - Parallel genetic algorithm completed in 38.8018 seconds.
  - Starting parallel PSO algorithm with 4 processes...
  - Parallel PSO algorithm completed in 39.4111 seconds.
Running for matrix size: 480
Starting run for matrix size 480 with 4 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 9.3823 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 18.9872 seconds.
  - Starting parallel genetic algorithm with 4 processes...
  - Parallel genetic algorithm completed in 40.1139 seconds.
  - Starting parallel PSO algorithm with 4 processes...
  - Parallel PSO algorithm completed in 40.3677 seconds.
Running for matrix size: 512
Starting run for matrix size 512 with 4 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 13.5925 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 27.5920 seconds.
  - Starting parallel genetic algorithm with 4 processes...
  - Parallel genetic algorithm completed in 42.3531 seconds.
  - Starting parallel PSO algorithm with 4 processes...
  - Parallel PSO algorithm completed in 41.6634 seconds.
```

```
Starting comparison of algorithms across different image sizes...
```

```
Running for matrix size: 120
Starting run for matrix size 120 with 16 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 0.2754 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 0.7055 seconds.
  - Starting parallel genetic algorithm with 16 processes...
  - Parallel genetic algorithm completed in 49.9150 seconds.
  - Starting parallel PSO algorithm with 16 processes...
  - Parallel PSO algorithm completed in 52.5241 seconds.
Running for matrix size: 240
Starting run for matrix size 240 with 16 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 2.1962 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 3.8179 seconds.
  - Starting parallel genetic algorithm with 16 processes...
```

```
  - Parallel genetic algorithm completed in 51.5110 seconds.
  - Starting parallel PSO algorithm with 16 processes...
  - Parallel PSO algorithm completed in 53.5832 seconds.
Running for matrix size: 360
Starting run for matrix size 360 with 16 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 6.0521 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 12.1577 seconds.
  - Starting parallel genetic algorithm with 16 processes...
  - Parallel genetic algorithm completed in 54.8246 seconds.
  - Starting parallel PSO algorithm with 16 processes...
  - Parallel PSO algorithm completed in 55.6844 seconds.
Running for matrix size: 480
Starting run for matrix size 480 with 16 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 9.5145 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 19.2747 seconds.
  - Starting parallel genetic algorithm with 16 processes...
  - Parallel genetic algorithm completed in 55.6263 seconds.
  - Starting parallel PSO algorithm with 16 processes...
  - Parallel PSO algorithm completed in 57.2177 seconds.
Running for matrix size: 512
Starting run for matrix size 512 with 16 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 13.7368 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 28.0355 seconds.
  - Starting parallel genetic algorithm with 16 processes...
  - Parallel genetic algorithm completed in 58.0518 seconds.
  - Starting parallel PSO algorithm with 16 processes...
  - Parallel PSO algorithm completed in 58.7831 seconds.
```

Zwiekszono populacje do 100

```
Starting comparison of algorithms across different image sizes...
Running for matrix size: 120
Starting run for matrix size 120 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 0.5155 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 1.3163 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 35.4095 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 36.0597 seconds.
Running for matrix size: 240
Starting run for matrix size 240 with 2 processes...
  - Starting sequential genetic algorithm...
```

```
  - Sequential genetic algorithm completed in 4.3160 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 7.4325 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 38.5903 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 39.1616 seconds.
Running for matrix size: 360
Starting run for matrix size 360 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 11.5744 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 23.2328 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 41.0582 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 42.1270 seconds.
Running for matrix size: 480
Starting run for matrix size 480 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 19.4041 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 38.4898 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 45.1577 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 45.9768 seconds.
Running for matrix size: 512
Starting run for matrix size 512 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 27.3253 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 54.6740 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 50.3200 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 50.1486 seconds.
```

Do 1000:

```
Starting comparison of algorithms across different image sizes...
```

```
Running for matrix size: 120
Starting run for matrix size 120 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 5.0681 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 12.6751 seconds.
  - Starting parallel genetic algorithm with 2 processes...
```

```
  - Parallel genetic algorithm completed in 38.8783 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 48.0434 seconds.
Running for matrix size: 240
Starting run for matrix size 240 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 43.1790 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 74.6257 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 62.1749 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 71.1949 seconds.
Running for matrix size: 360
Starting run for matrix size 360 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 112.9713 seconds.
  - Starting sequential PSO algorithm...
  - Sequential PSO algorithm completed in 231.4169 seconds.
  - Starting parallel genetic algorithm with 2 processes...
  - Parallel genetic algorithm completed in 94.8493 seconds.
  - Starting parallel PSO algorithm with 2 processes...
  - Parallel PSO algorithm completed in 102.4244 seconds.
Running for matrix size: 480
Starting run for matrix size 480 with 2 processes...
  - Starting sequential genetic algorithm...
  - Sequential genetic algorithm completed in 189.3037 seconds.
  - Starting sequential PSO algorithm...
```

Dla rozmiaru obrazu 360 pierwsze zyski urownoleglenia.

```
  - Starting sequential PSO algorithm
...Starting run for matrix size 360 with 4 processes... -
 Starting sequential genetic algorithm... -
Sequential genetic algorithm completed in 116.3955 seconds. -
Starting sequential PSO algorithm... -
 Sequential PSO algorithm completed in 237.6956 seconds. -
 Starting parallel genetic algorithm with 4 processes... -
 Parallel genetic algorithm completed in 71.9046 seconds. -
Starting parallel PSO algorithm with 4 processes... -
 Parallel PSO algorithm completed in 78.4662 seconds.
```

Po testach zapisu pliku time_seq vs. time_par

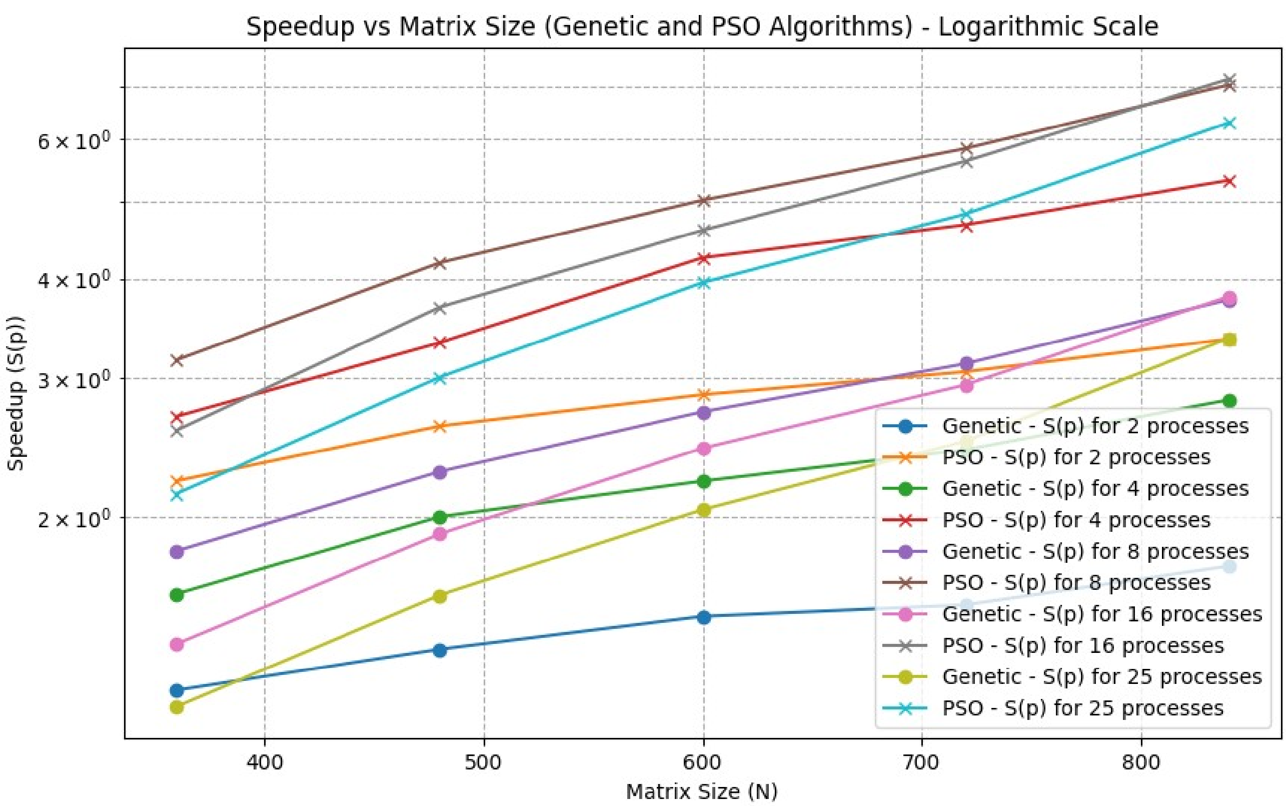
Dodano skrypt testowy dla nastepujacych danych w celu obliczenia wydajnosci:

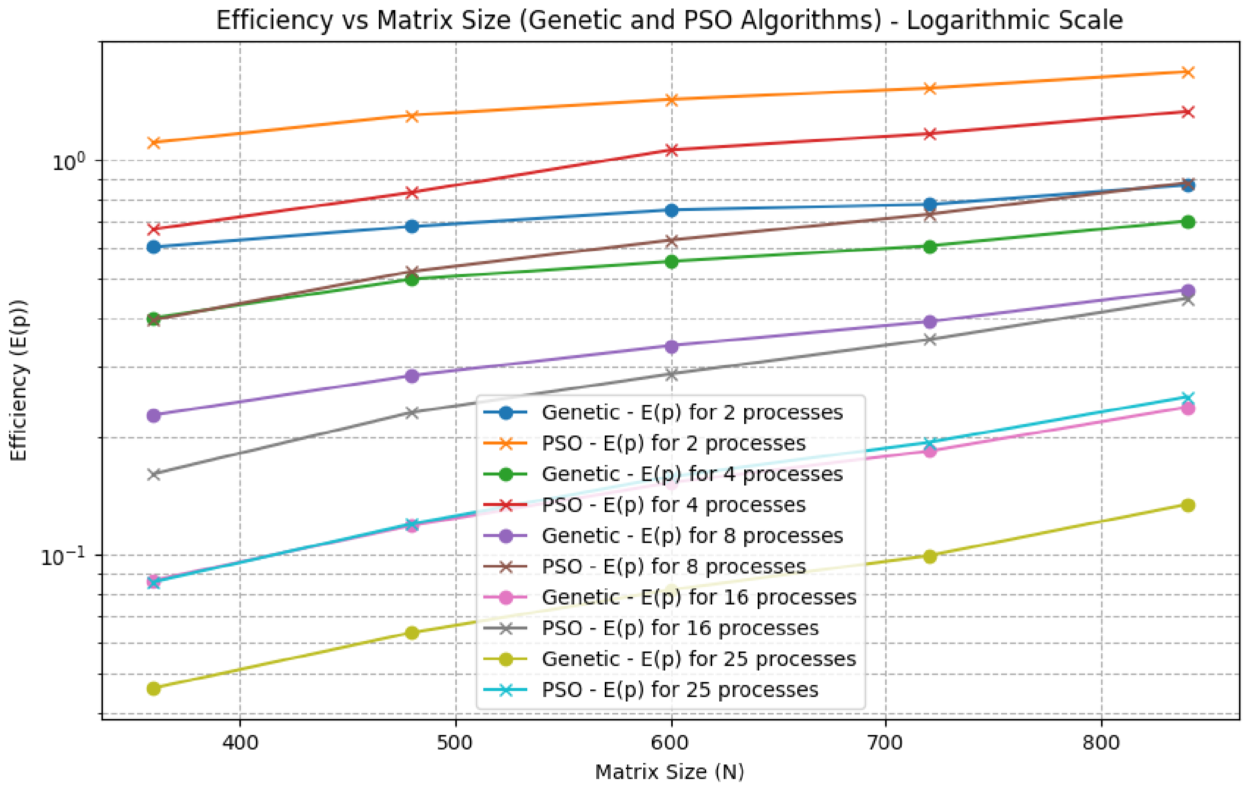
Rozmiary obrazu: [360, 480, 512, 640, 720]

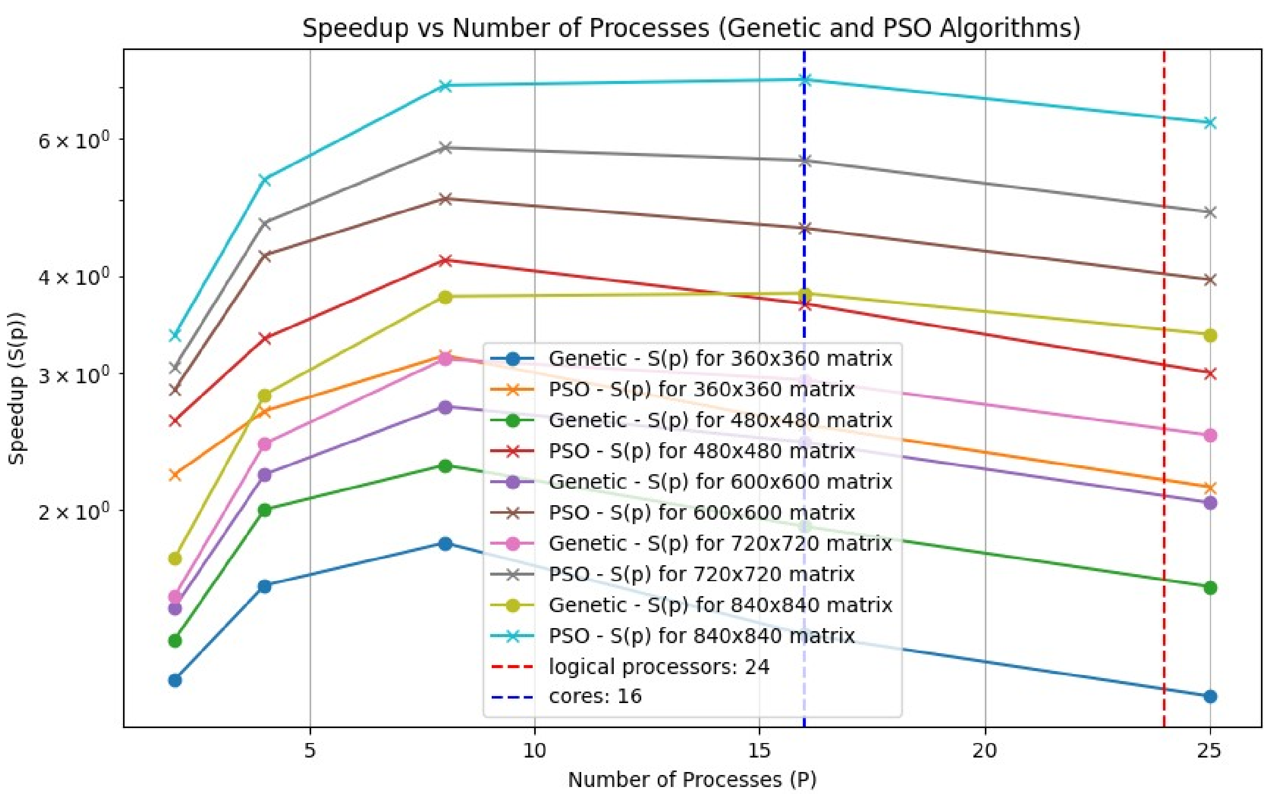
Liczba procesow: [2, 4, 8, 16, 25]

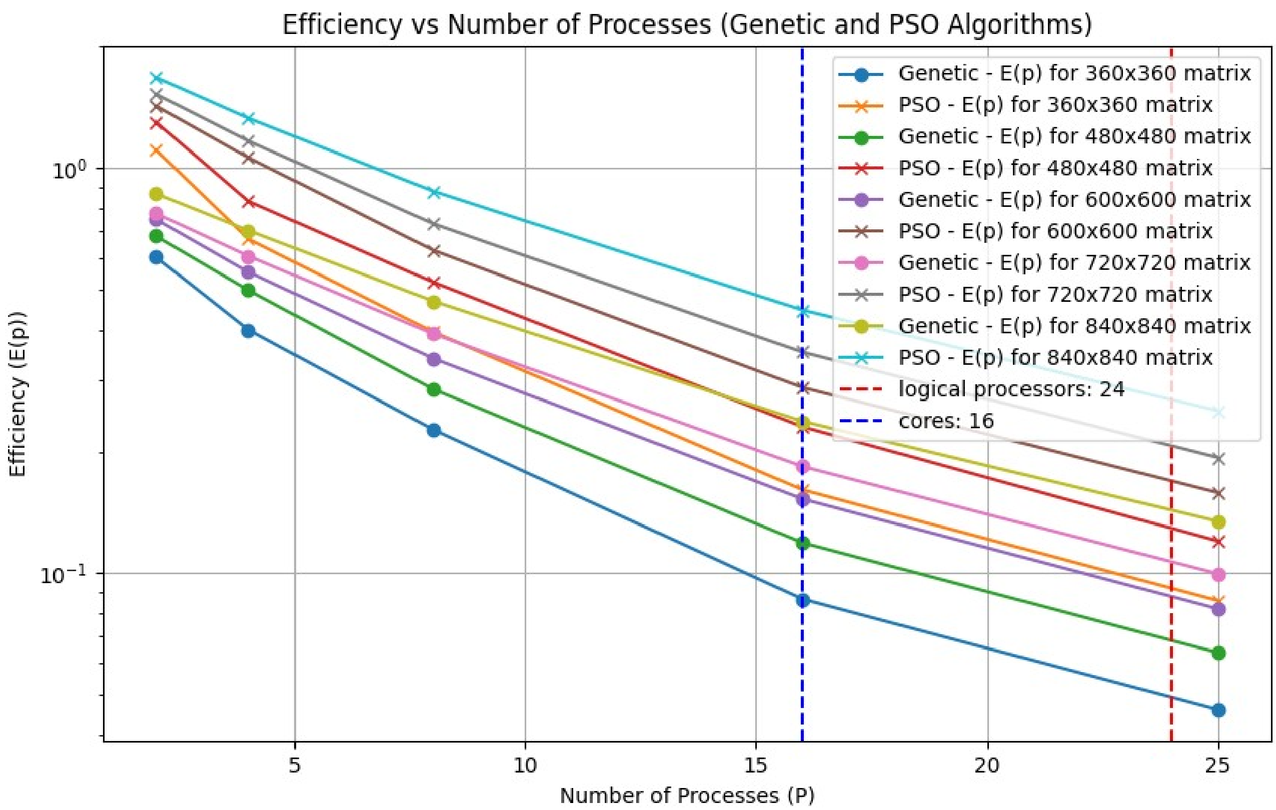
Populacje : 1000

Epoki : 100




Efficiency vs Matrix Size (Genetic and PSO Algorithms) - Logarithmic Scale
Efficiency (E(p))
Matrix Size (N)
10^0
10^{-1}
400
500
600
700
800
Genetic - E(p) for 2 processes
PSO - E(p) for 2 processes
Genetic - E(p) for 4 processes
PSO - E(p) for 4 processes
Genetic - E(p) for 8 processes
PSO - E(p) for 8 processes
Genetic - E(p) for 16 processes
PSO - E(p) for 16 processes
Genetic - E(p) for 25 processes
PSO - E(p) for 25 processes


Speedup vs Number of Processes (Genetic and PSO Algorithms)
Speedup (S(p))
Number of Processes (P)
6×10^0
4×10^0
3×10^0
2×10^0
5
10
15
20
25
Genetic - S(p) for 360x360 matrix
PSO - S(p) for 360x360 matrix
Genetic - S(p) for 480x480 matrix
PSO - S(p) for 480x480 matrix
Genetic - S(p) for 600x600 matrix
PSO - S(p) for 600x600 matrix
Genetic - S(p) for 720x720 matrix
PSO - S(p) for 720x720 matrix
Genetic - S(p) for 840x840 matrix
PSO - S(p) for 840x840 matrix
logical processors: 24
cores: 16



Efficiency vs Number of Processes (Genetic and PSO Algorithms)
Genetic - E(p) for 360x360 matrix
PSO - E(p) for 360x360 matrix
Genetic - E(p) for 480x480 matrix
PSO - E(p) for 480x480 matrix
Genetic - E(p) for 600x600 matrix
PSO - E(p) for 600x600 matrix
Genetic - E(p) for 720x720 matrix
PSO - E(p) for 720x720 matrix
Genetic - E(p) for 840x840 matrix
PSO - E(p) for 840x840 matrix
logical processors: 24
cores: 16
Efficiency (E(p))
10^0
10^{-1}
5
10
15
20
25
Number of Processes (P)

BACKUP CMD:

Starting comparison of algorithms across different image sizes...

Running sequential algorithms for matrix size: 360

Starting sequential run for matrix size 360...

- Starting sequential genetic algorithm...

- Sequential genetic algorithm completed in 114.8637 seconds.

- Starting sequential PSO algorithm...

- Sequential PSO algorithm completed in 231.9453 seconds.

Running sequential algorithms for matrix size: 480

Starting sequential run for matrix size 480...

- Starting sequential genetic algorithm...

- Sequential genetic algorithm completed in 183.8182 seconds.

- Starting sequential PSO algorithm...

- Sequential PSO algorithm completed in 373.4458 seconds.

Running sequential algorithms for matrix size: 600

Starting sequential run for matrix size 600...

- Starting sequential genetic algorithm...

- Sequential genetic algorithm completed in 275.7816 seconds.

- Starting sequential PSO algorithm...

- Sequential PSO algorithm completed in 554.8547 seconds.

Running sequential algorithms for matrix size: 720

Starting sequential run for matrix size 720...

- Starting sequential genetic algorithm...

- Sequential genetic algorithm completed in 387.9285 seconds.

- Starting sequential PSO algorithm...

- Sequential PSO algorithm completed in 789.9013 seconds.

Running sequential algorithms for matrix size: 840

Starting sequential run for matrix size 840...

- Starting sequential genetic algorithm...

- Sequential genetic algorithm completed in 602.4391 seconds.

- Starting sequential PSO algorithm...

- Sequential PSO algorithm completed in 1194.8779 seconds.

Running for matrix size: 360 and num_processes: 2

Starting parallel run for matrix size 360 with 2 processes...

- Starting parallel genetic algorithm with 2 processes...

- Parallel genetic algorithm completed in 95.0529 seconds.

- Starting parallel PSO algorithm with 2 processes...

- Parallel PSO algorithm completed in 104.4107 seconds.

Running for matrix size: 480 and num_processes: 2

Starting parallel run for matrix size 480 with 2 processes...

- Starting parallel genetic algorithm with 2 processes...

- Parallel genetic algorithm completed in 135.0131 seconds.

- Starting parallel PSO algorithm with 2 processes...

- Parallel PSO algorithm completed in 143.3092 seconds.

Running for matrix size: 600 and num_processes: 2

Starting parallel run for matrix size 600 with 2 processes...

- Starting parallel genetic algorithm with 2 processes...

- Parallel genetic algorithm completed in 184.0997 seconds.

- Starting parallel PSO algorithm with 2 processes...

- Parallel PSO algorithm completed in 194.3363 seconds.

Running for matrix size: 720 and num_processes: 2

Starting parallel run for matrix size 720 with 2 processes...

- Starting parallel genetic algorithm with 2 processes...

- Parallel genetic algorithm completed in 250.4942 seconds.

- Starting parallel PSO algorithm with 2 processes...

- Parallel PSO algorithm completed in 258.8205 seconds.

Running for matrix size: 840 and num_processes: 2

Starting parallel run for matrix size 840 with 2 processes...

- Starting parallel genetic algorithm with 2 processes...

- Parallel genetic algorithm completed in 347.6599 seconds.

- Starting parallel PSO algorithm with 2 processes...

- Parallel PSO algorithm completed in 356.0165 seconds.

Results saved to result_2.txt

Running for matrix size: 360 and num_processes: 4

Starting parallel run for matrix size 360 with 4 processes...

- Starting parallel genetic algorithm with 4 processes...

- Parallel genetic algorithm completed in 71.8516 seconds.

- Starting parallel PSO algorithm with 4 processes...

- Parallel PSO algorithm completed in 86.5523 seconds.

Running for matrix size: 480 and num_processes: 4

Starting parallel run for matrix size 480 with 4 processes...

- Starting parallel genetic algorithm with 4 processes...

- Parallel genetic algorithm completed in 91.8388 seconds.

- Starting parallel PSO algorithm with 4 processes...

- Parallel PSO algorithm completed in 112.4498 seconds.

Running for matrix size: 600 and num_processes: 4

Starting parallel run for matrix size 600 with 4 processes...

- Starting parallel genetic algorithm with 4 processes...

- Parallel genetic algorithm completed in 124.2159 seconds.

- Starting parallel PSO algorithm with 4 processes...

- Parallel PSO algorithm completed in 130.4708 seconds.

Running for matrix size: 720 and num_processes: 4

Starting parallel run for matrix size 720 with 4 processes...

- Starting parallel genetic algorithm with 4 processes...

- Parallel genetic algorithm completed in 159.4746 seconds.

- Starting parallel PSO algorithm with 4 processes...

- Parallel PSO algorithm completed in 168.7484 seconds.

Running for matrix size: 840 and num_processes: 4

Starting parallel run for matrix size 840 with 4 processes...

- Starting parallel genetic algorithm with 4 processes...

- Parallel genetic algorithm completed in 214.2276 seconds.

- Starting parallel PSO algorithm with 4 processes...

- Parallel PSO algorithm completed in 224.3370 seconds.

Results saved to result_4.txt

Running for matrix size: 360 and num_processes: 8

Starting parallel run for matrix size 360 with 8 processes...

- Starting parallel genetic algorithm with 8 processes...

- Parallel genetic algorithm completed in 63.3691 seconds.

- Starting parallel PSO algorithm with 8 processes...

- Parallel PSO algorithm completed in 73.4422 seconds.

Running for matrix size: 480 and num_processes: 8

Starting parallel run for matrix size 480 with 8 processes...

- Starting parallel genetic algorithm with 8 processes...

- Parallel genetic algorithm completed in 80.5510 seconds.

- Starting parallel PSO algorithm with 8 processes...

- Parallel PSO algorithm completed in 89.1278 seconds.

Running for matrix size: 600 and num_processes: 8

Starting parallel run for matrix size 600 with 8 processes...

- Starting parallel genetic algorithm with 8 processes...

- Parallel genetic algorithm completed in 101.5211 seconds.

- Starting parallel PSO algorithm with 8 processes...

- Parallel PSO algorithm completed in 110.3614 seconds.

Running for matrix size: 720 and num_processes: 8

Starting parallel run for matrix size 720 with 8 processes...

- Starting parallel genetic algorithm with 8 processes...

- Parallel genetic algorithm completed in 124.1439 seconds.

- Starting parallel PSO algorithm with 8 processes...

- Parallel PSO algorithm completed in 135.1624 seconds.

Running for matrix size: 840 and num_processes: 8

Starting parallel run for matrix size 840 with 8 processes...

- Starting parallel genetic algorithm with 8 processes...

- Parallel genetic algorithm completed in 160.0736 seconds.

- Starting parallel PSO algorithm with 8 processes...

- Parallel PSO algorithm completed in 169.9084 seconds.

Results saved to result_8.txt

Running for matrix size: 360 and num_processes: 16

Starting parallel run for matrix size 360 with 16 processes...

- Starting parallel genetic algorithm with 16 processes...

- Parallel genetic algorithm completed in 83.0929 seconds.

- Starting parallel PSO algorithm with 16 processes...
- Parallel PSO algorithm completed in 90.1175 seconds.

Running for matrix size: 480 and num_processes: 16

Starting parallel run for matrix size 480 with 16 processes...

- Starting parallel genetic algorithm with 16 processes...
- Parallel genetic algorithm completed in 96.5525 seconds.
- Starting parallel PSO algorithm with 16 processes...
- Parallel PSO algorithm completed in 101.4189 seconds.

Running for matrix size: 600 and num_processes: 16

Starting parallel run for matrix size 600 with 16 processes...

- Starting parallel genetic algorithm with 16 processes...
- Parallel genetic algorithm completed in 112.8883 seconds.
- Starting parallel PSO algorithm with 16 processes...
- Parallel PSO algorithm completed in 120.4560 seconds.

Running for matrix size: 720 and num_processes: 16

Starting parallel run for matrix size 720 with 16 processes...

- Starting parallel genetic algorithm with 16 processes...
- Parallel genetic algorithm completed in 132.0229 seconds.
- Starting parallel PSO algorithm with 16 processes...
- Parallel PSO algorithm completed in 140.2886 seconds.

Running for matrix size: 840 and num_processes: 16

Starting parallel run for matrix size 840 with 16 processes...

- Starting parallel genetic algorithm with 16 processes...
- Parallel genetic algorithm completed in 158.6276 seconds.
- Starting parallel PSO algorithm with 16 processes...
- Parallel PSO algorithm completed in 167.0454 seconds.

Results saved to result_16.txt

Running for matrix size: 360 and num_processes: 25

Starting parallel run for matrix size 360 with 25 processes...

- Starting parallel genetic algorithm with 25 processes...

- Parallel genetic algorithm completed in 99.6992 seconds.

- Starting parallel PSO algorithm with 25 processes...

- Parallel PSO algorithm completed in 108.3729 seconds.

Running for matrix size: 480 and num_processes: 25

Starting parallel run for matrix size 480 with 25 processes...

- Starting parallel genetic algorithm with 25 processes...

- Parallel genetic algorithm completed in 115.4085 seconds.

- Starting parallel PSO algorithm with 25 processes...

- Parallel PSO algorithm completed in 124.3585 seconds.

Running for matrix size: 600 and num_processes: 25

Starting parallel run for matrix size 600 with 25 processes...

- Starting parallel genetic algorithm with 25 processes...

- Parallel genetic algorithm completed in 134.9761 seconds.

- Starting parallel PSO algorithm with 25 processes...

- Parallel PSO algorithm completed in 140.2825 seconds.

Running for matrix size: 720 and num_processes: 25

Starting parallel run for matrix size 720 with 25 processes...

- Starting parallel genetic algorithm with 25 processes...

- Parallel genetic algorithm completed in 155.5092 seconds.

- Starting parallel PSO algorithm with 25 processes...

- Parallel PSO algorithm completed in 163.5454 seconds.

Running for matrix size: 840 and num_processes: 25

Starting parallel run for matrix size 840 with 25 processes...

- Starting parallel genetic algorithm with 25 processes...

- Parallel genetic algorithm completed in 179.1087 seconds.

- Starting parallel PSO algorithm with 25 processes...

- Parallel PSO algorithm completed in 189.7299 seconds.

Results saved to result_25.txt

ANALIZA TEORETYCZNA

```
def create_chromosome_array():
    return np.random.choice([0, 1], size=(21,))
```

O(1)

```
def hide_bit(value, bit):
    result = np.uint8((value & 0xFE) | bit)

    return result
```

O(1)

```
def get_traversal(start_h, start_w, direction, height, width):
    direction_code = direction[0] * 4 + direction[1] * 2 + direction[2]
    if direction_code == 0:
        return range(start_h, height), range(start_w, width)
    elif direction_code == 1:  # Right-Down
        return range(start_w, width), range(start_h, height)
    elif direction_code == 2:  # Down-Left
        return range(start_h, height), range(width - 1 - start_w, -1, -1)
    elif direction_code == 3:  # Left-Down
        return range(width - 1 - start_w, -1, -1), range(start_h, height)
    elif direction_code == 4:  # Left-Up
        return range(width - 1 - start_w, -1, -1), range(height - 1 - start_h, -1, -1)
    elif direction_code == 5:  # Up-Left
        return range(height - 1 - start_h, -1, -1), range(width - 1 - start_w, -1, -1)
    elif direction_code == 6:  # Up-Right
        return range(height - 1 - start_h, -1, -1), range(start_w, width)
    elif direction_code == 7:  # Right-Up
        return range(start_w, width), range(height - 1 - start_h, -1, -1)
```

O(1)

```python
def Encode(host, binary_message, binary_chromosome, delimiter):

    height, width = host.shape
    transformed = host.copy()
    start_h = int(''.join(map(str, binary_chromosome[2:10])), 2)
    start_w = int(''.join(map(str, binary_chromosome[10:18])), 2)
    direction = binary_chromosome[18:21]
    rows, cols = get_traversal(start_h, start_w, direction, height, width)
    message_idx, delimiter_idx = 0, 0
    for h in rows:
        for w in cols:
            if message_idx < len(binary_message):
                transformed[h, w] = hide_bit(
                    transformed[h, w], binary_message[message_idx])
                message_idx += 1
            elif delimiter_idx < len(delimiter):
                transformed[h, w] = hide_bit(
                    transformed[h, w], delimiter[delimiter_idx])
                delimiter_idx += 1
            else:
                return transformed
    return transformed
```

O(width * height) = O(N^2(

```python
def calculate_BER(original, received):

    if len(original) != len(received):
        raise ValueError("Both binary sequences must have the same length.")
    bit_errors = np.count_nonzero(original != received)
    return bit_errors / original.size
```

np.count_nonzero – O(N^2)

```python
def fitness_function_array(args):

    #print(f"args: {args}")
    chromosome, image, binary_data, delimiter = args
    secret = Encode(image, binary_data, chromosome, delimiter)
    original_binary_data = np.unpackbits(image.astype(np.uint8))
    secret_binary_data = np.unpackbits(secret.astype(np.uint8))
    psnr = cv2.PSNR(image, secret)
    ber = calculate_BER(original_binary_data, secret_binary_data)
    #print("COmplete")
    return psnr, ber
```

cv2.PSNR – O(N^2)

np.unpackbits - O(N^2)

OGOLNE : O(N^2)

```
def parallel_fitness(population, image, binary_data, delimiter, num_processes):

    with Pool(processes=num_processes) as pool:
        fitness_scores = pool.map(fitness_function_array,
                                  [(chromosome, image, binary_data, delimiter) for chromosome in population])
    return fitness_scores
```

O(wielkosc_populacji * N^2/liczba_procesow)

ROWNOLEGLE

```
def genetic_with_tracking_array(image, binary_data, delimiter, num_processes):

    population_size = 1000
    num_generations = 100
    mutation_rate = 0.3
    num_bits = 21
    population = [create_chromosome_array() for _ in range(population_size)]

    psnr_values = []
    ber_values = []

    global_best_position = population[0]
    global_best_score = float('-inf')

    for generation in range(num_generations):
        fitness_scores = parallel_fitness(
            population, image, binary_data, delimiter, num_processes)
        psnr_scores = [score[0] for score in fitness_scores]
        ber_scores = [score[1] for score in fitness_scores]

        max_psnr_index = np.argmax(psnr_scores)
        max_psnr_value = psnr_scores[max_psnr_index]
        best_chromosome = population[max_psnr_index]

        if max_psnr_value > global_best_score:
            global_best_score = max_psnr_value
            global_best_position = best_chromosome

        psnr_values.append(global_best_score)

        ber_values.append(ber_scores[max_psnr_index])

        selected_parents = random.choices(
            population, weights=psnr_scores, k=population_size)

        offspring = []
        for i in range(0, population_size, 2):
            parent1 = selected_parents[i]
```

```python
        parent2 = selected_parents[i + 1]
        crossover_point = np.random.randint(1, num_bits - 1)
        offspring1 = np.concatenate(
            [parent1[:crossover_point], parent2[crossover_point:]])
        offspring2 = np.concatenate(
            [parent2[:crossover_point], parent1[crossover_point:]])
        offspring.extend([offspring1, offspring2])

    for i in range(population_size):
        for j in range(num_bits):
            if np.random.random() < mutation_rate:
                offspring[i][j] = 1 - offspring[i][j]

    offspring[-1] = global_best_position
    population = offspring

  return psnr_values, ber_values
```

O(populacja * generacje * N^2/liczba procesow)

```python
def pso_with_tracking_array(image, binary_data, delimiter, num_processes):

  num_particles = 1000
  num_dimensions = 21
  max_iterations = 100
  c1, c2 = 2.0, 2.0  # Cognitive and social coefficients
  w_max, w_min = 0.9, 0.4  # Inertia weight range

  particles = [create_chromosome_array() for _ in range(num_particles)]
  velocities = np.random.uniform(-1, 1, (num_particles, num_dimensions))

  personal_best_positions = particles.copy()
  personal_best_scores = [float('-inf')] * num_particles

  global_best_position = particles[0]
  global_best_score = float('-inf')

  psnr_values = []
  ber_values = []

  for iteration in range(max_iterations):
    fitness_scores = parallel_fitness(
        particles, image, binary_data, delimiter, num_processes)
    psnr_scores = [score[0] for score in fitness_scores]
    ber_scores = [score[1] for score in fitness_scores]

    for i, score in enumerate(psnr_scores):
        if score > personal_best_scores[i]:
            personal_best_scores[i] = score
```

```
                personal_best_positions[i] = particles[i].copy()
            if score > global_best_score:
                global_best_score = score
                global_best_position = particles[i].copy()

        psnr_values.append(global_best_score)
        ber_values.append(ber_scores[np.argmax(psnr_scores)])

        w = w_max - (w_max - w_min) * (iteration / max_iterations)

        for i in range(num_particles):
            for j in range(num_dimensions):
                r1, r2 = random.random(), random.random()
                cognitive = c1 * r1 * (personal_best_positions[i][j] - particles[i][j])
                social = c2 * r2 * (global_best_position[j] - particles[i][j])
                velocities[i][j] = w * velocities[i][j] + cognitive + social

                velocities[i][j] = np.clip(velocities[i][j], -4, 4)

                probability = 1 / (1 + np.exp(-velocities[i][j]))
                if random.random() < probability:
                    particles[i][j] = 1 - particles[i][j]

    return psnr_values, ber_values
```

O(populacja * generacje * N^2/liczba procesow)

SEKEWNCYJNE

```
def genetic_with_tracking_array_seq(image, binary_data, delimiter):

    global best_solution
    population_size = 1000
    num_generations = 100
    mutation_rate = 0.3
    num_bits = 21
    population = [create_chromosome_array() for _ in range(population_size)]
    psnr_values = []
    ber_values = []

    best_individual = population[0]

    for generation in range(num_generations):
        fitness_scores = [fitness_function_array(
            (chromosome, image, binary_data, delimiter))[0] for chromosome in population]
        psnr_values.append(max(fitness_scores))
        best_individual = population[np.argmax(fitness_scores)]
        ber_values.append(fitness_function_array(
            (best_individual, image, binary_data, delimiter))[1])
```

```
        selected_parents = random.choices(
            population, weights=fitness_scores, k=population_size)

        offspring = []
        for i in range(0, population_size, 2):
            parent1 = selected_parents[i]
            parent2 = selected_parents[i + 1]
            crossover_point = np.random.randint(1, num_bits - 1)
            offspring1 = np.concatenate(
                [parent1[:crossover_point], parent2[crossover_point:]])
            offspring2 = np.concatenate(
                [parent2[:crossover_point], parent1[crossover_point:]])
            offspring.extend([offspring1, offspring2])

        for i in range(population_size):
            for j in range(num_bits):
                if np.random.random() < mutation_rate:
                    offspring[i][j] = 1 - offspring[i][j]

        best_fitness_index = np.argmax(fitness_scores)
        best_chromosome = population[best_fitness_index]

        offspring[-1] = best_chromosome
        population = offspring

    best_solution = max(population, key=lambda x: fitness_function_array(
        (x, image, binary_data, delimiter))[0])
    return psnr_values, ber_values
```

O(N^2 * populacja * generacje)

```
def pso_with_tracking_array_seq(image, binary_data, delimiter):
    num_particles = 1000
    num_dimensions = 21
    max_iterations = 100
    c1, c2 = 2.0, 2.0
    global_best_position = None

    particles = [create_chromosome_array() for _ in range(num_particles)]
    velocities = [[random.uniform(-1, 1) for _ in range(num_dimensions)]
                  for _ in range(num_particles)]
    best_positions = particles[:]
    best_fitness = [fitness_function_array((p, image, binary_data, delimiter))[0]
                    for p in particles]

    psnr_values = []
    ber_values = []
```

```python
    fitness_scores = [fitness_function_array((p, image, binary_data, delimiter))[0]
                      for p in particles]
    global_best_position = particles[np.argmax(fitness_scores)]

    for iteration in range(max_iterations):
        for i in range(num_particles):
            fitness = fitness_function_array(
                (particles[i], image, binary_data, delimiter))[0]

            if fitness > best_fitness[i]:
                best_fitness[i] = fitness
                best_positions[i] = particles[i].copy()

            if global_best_position is None or fitness > fitness_function_array(
                    (global_best_position, image, binary_data, delimiter))[0]:
                global_best_position = particles[i].copy()

        psnr_values.append(fitness_function_array(
            (global_best_position, image, binary_data, delimiter))[0])
        ber_values.append(fitness_function_array(
            (global_best_position, image, binary_data, delimiter))[1])

        w = 0.9 - (0.5 * iteration / max_iterations)

        for i in range(num_particles):
            for j in range(num_dimensions):
                r1, r2 = random.random(), random.random()

                cognitive = c1 * r1 * \
                    (int(best_positions[i][j]) - int(particles[i][j]))
                social = c2 * r2 * \
                    (int(global_best_position[j]) - int(particles[i][j]))

                velocities[i][j] = w * velocities[i][j] + cognitive + social

                velocities[i][j] = np.clip(velocities[i][j], -4, 4)

                # Sigmoid-based update
                probability = 1 / (1 + np.exp(-velocities[i][j]))
                if np.random.random() < probability:
                    particles[i][j] = 1 - particles[i][j]  # Flip the bit

    return psnr_values, ber_values
```

O(N^2 * populacja * generacje)

```python
def run_sequential_for_size(image, delimiter, matrix_size):

    print(f"Starting sequential run for matrix size {matrix_size}...")

    binary_data = np.random.randint(
        0, 2, math.ceil(matrix_size * 3/5), dtype=np.uint8)
    cropped_image = cv2.resize(
        image, (matrix_size, matrix_size), interpolation=cv2.INTER_LINEAR)

    print(f"  - Starting sequential genetic algorithm...")
    start_time = time.time()
    psnr_gen_seq, ber_gen_seq = genetic_with_tracking_array_seq(
        cropped_image, binary_data, delimiter)
    end_time = time.time()
    sequential_time_gen = end_time - start_time
    print(
        f"  - Sequential genetic algorithm completed in {sequential_time_gen:.4f} seconds.")

    print(f"  - Starting sequential PSO algorithm...")
    start_time = time.time()
    psnr_gen_seq, ber_gen_seq = pso_with_tracking_array_seq(
        cropped_image, binary_data, delimiter)
    end_time = time.time()
    sequential_time_pso = end_time - start_time
    print(
        f"  - Sequential PSO algorithm completed in {sequential_time_pso:.4f} seconds.")

    return sequential_time_gen, sequential_time_pso
```

O(binary_message * matrix_size^2 + 2(generacje*populacje)

```python
def run_parallel_for_size(image, delimiter, matrix_size, num_processes, sequential_times):
    print(f"Starting parallel run for matrix size {matrix_size} with {num_processes} processes...")

    binary_data = np.random.randint(
        0, 2, math.ceil(matrix_size * 3/5), dtype=np.uint8)
    cropped_image = cv2.resize(
        image, (matrix_size, matrix_size), interpolation=cv2.INTER_LINEAR)

    print(f"  - Starting parallel genetic algorithm with {num_processes} processes...")
    start_time = time.time()
    psnr_gen_par, ber_gen_par = genetic_with_tracking_array(
        cropped_image, binary_data, delimiter, num_processes)
    end_time = time.time()
    parallel_time_gen = end_time - start_time
    print(
        f"  - Parallel genetic algorithm completed in {parallel_time_gen:.4f} seconds.")

    print(f"  - Starting parallel PSO algorithm with {num_processes} processes...")
    start_time = time.time()
    psnr_gen_par, ber_gen_par = pso_with_tracking_array(
```

```
        cropped_image, binary_data, delimiter, num_processes)
    end_time = time.time()
    parallel_time_pso = end_time - start_time
    print(
        f"  - Parallel PSO algorithm completed in {parallel_time_pso:.4f} seconds.")

    return matrix_size, num_processes, sequential_times[0], parallel_time_gen, sequential_times[1], 
parallel_time_pso
```

O(binary_message * matrix_size^2 + (2(generacje*populacje)/liczba procesow)

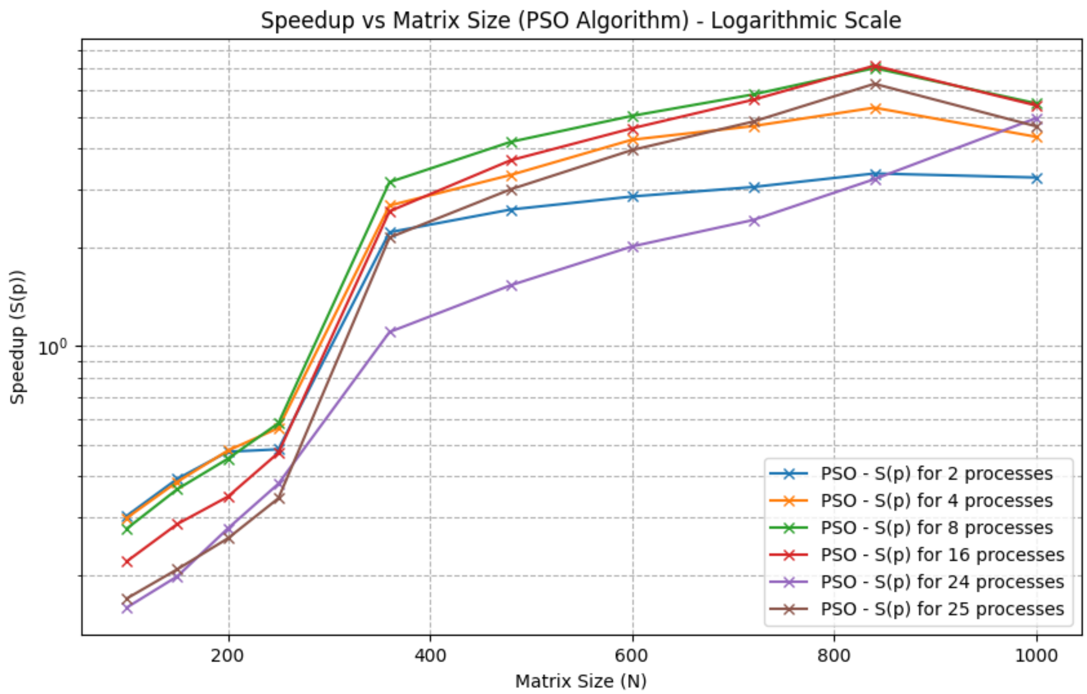
```
def compare_algorithms(image, delimiter, matrix_sizes, num_processes):
    print(f"Starting comparison of algorithms across different image sizes...")
    sequential_results = {}
    for matrix_size in matrix_sizes:
        print(f"Running sequential algorithms for matrix size: {matrix_size}")
        sequential_times = run_sequential_for_size(
            image, delimiter, matrix_size)
        sequential_results[matrix_size] = sequential_times

    for num_proc in num_processes:
        csv_filename = f"result_{num_proc}.txt"
        with open(csv_filename, mode='w', newline='') as file:
            writer = csv.writer(file)
            writer.writerow(['MatrixSize', 'NumProcesses', 'SequentialTime_gen', 'ParallelTime_gen', 
'SequentialTime_pso', 'ParallelTime_pso'])  # Header

            for matrix_size in matrix_sizes:
                print(f"Running for matrix size: {matrix_size} and num_processes: {num_proc}")
                sequential_time_gen, sequential_time_pso = sequential_results[matrix_size]
                result = run_parallel_for_size(
                    image, delimiter, matrix_size, num_proc, (sequential_time_gen, sequential_time_pso))
                writer.writerow(result)

        print(f"Results saved to {csv_filename}")
```

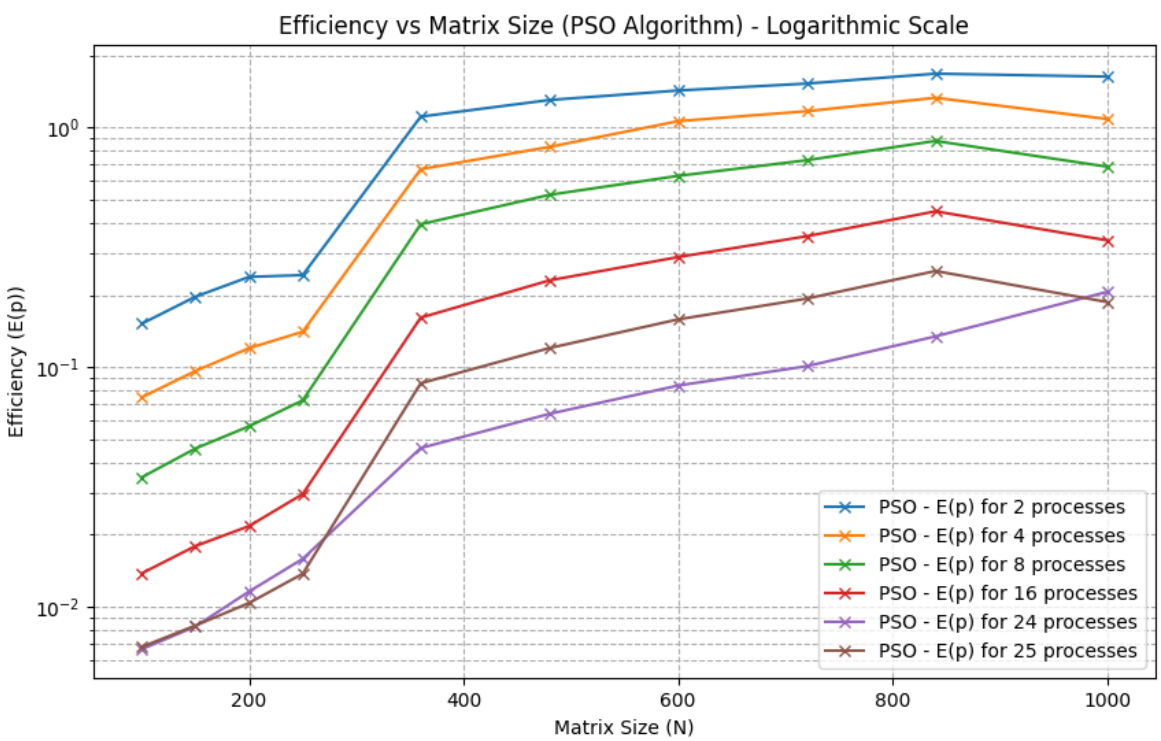
O((liczba_macierzy * liczba procesow * (O_ run_parallel_for_size)) + liczba_macierzy * (O_ run_sequential_for_size)

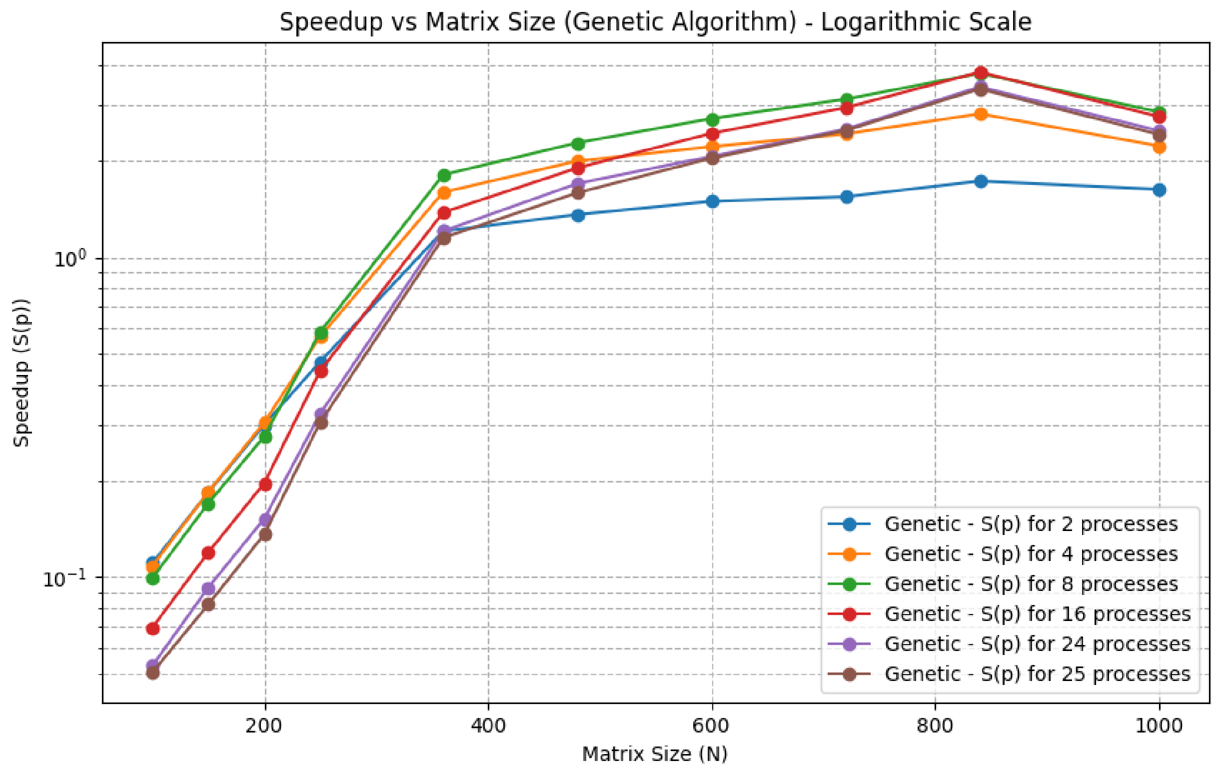
PROFILER:

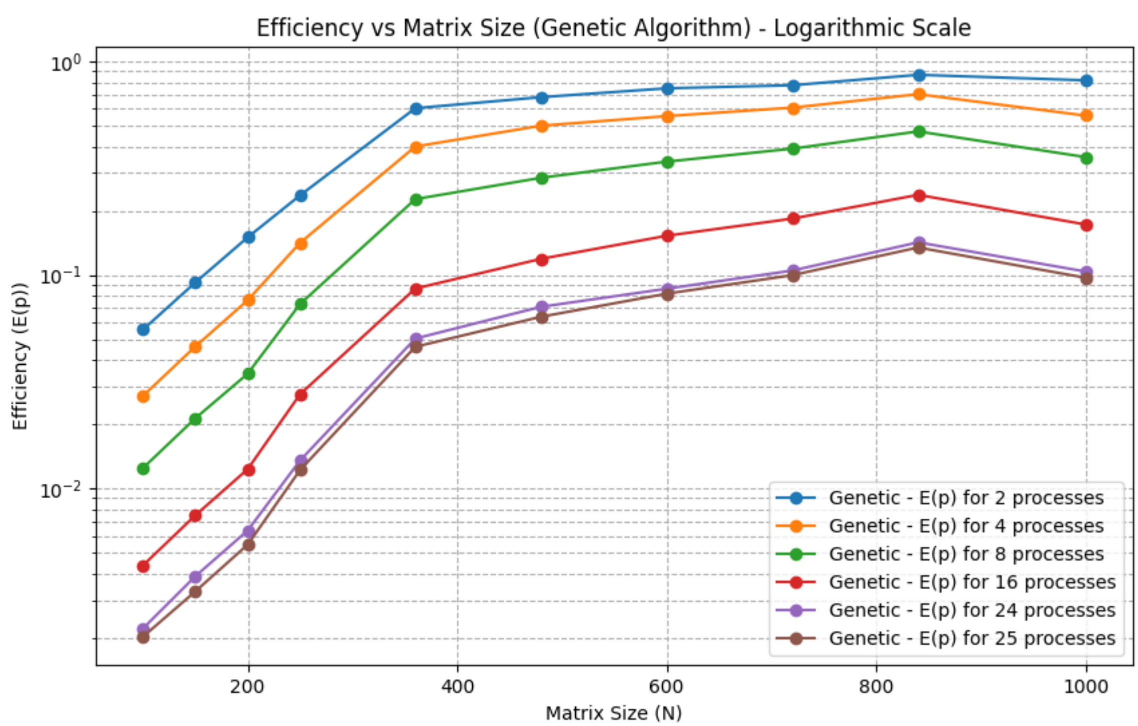
```
ncalls  tottime  percall  cumtime  percall filename:lineno(function)

        1    0.001    0.001 7802.050 7802.050 ProjektPrzejsciowy.py:473(compare_algorithms)
317322/317308    2.303    0.000 6455.609    0.020 connection.py:246(recv)
   389000    9.317    0.000 6094.872    0.016 connection.py:202(send)
419000/418722    3.172    0.000 6082.487    0.015 connection.py:284(_send_bytes)
       30    0.009    0.000 6007.590  200.253 ProjektPrzejsciowy.py:444(run_parallel_for_size)
     6000    0.014    0.000 5530.466    0.922 ProjektPrzejsciowy.py:108(parallel_fitness)
```

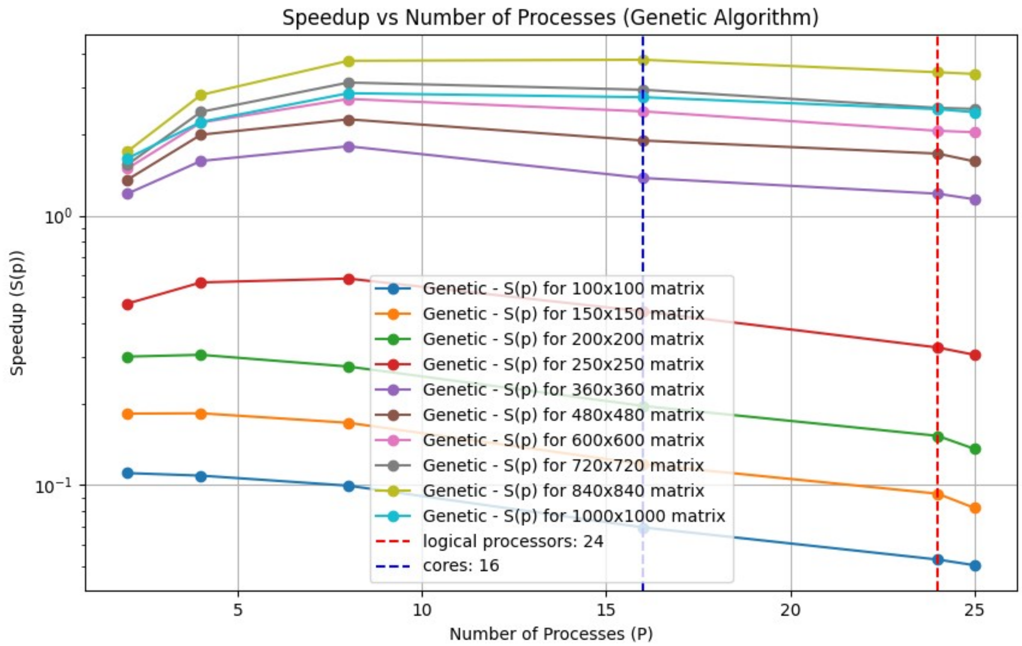
```
     6000    0.054    0.000 5530.452    0.922 pool.py:738(__exit__)
     6000    0.084    0.000 5530.363    0.922 pool.py:654(terminate)
    85001    0.158    0.000 5361.570    0.063 util.py:208(__call__)
     6000    0.311    0.000 5358.774    0.893 pool.py:680(_terminate_pool)
18000/6000    2.486    0.000 5257.711    0.876 threading.py:1018(_bootstrap)
18000/6000    0.322    0.000 5257.157    0.876 threading.py:1058(_bootstrap_inner)
18000/6000    0.120    0.000 5079.967    0.847 threading.py:1001(run)
     6000    0.043    0.000 4918.949    0.820 pool.py:671(_help_stuff_finish)
   227707    2.054    0.000 4792.837    0.021 pool.py:500(_wait_for_updates)
     6000    0.125    0.000 4754.428    0.792 pool.py:573(_handle_results)
     6000   18.385    0.003 4398.900    0.733 {method 'acquire' of '_multiprocessing.SemLock' objects}
328423/317564    2.896    0.000 4198.662    0.013 connection.py:310(_recv_bytes)
     6000    0.091    0.000 4112.540    0.685 pool.py:527(_handle_tasks)
     6000    0.267    0.000 4053.556    0.676 pool.py:362(map)
```

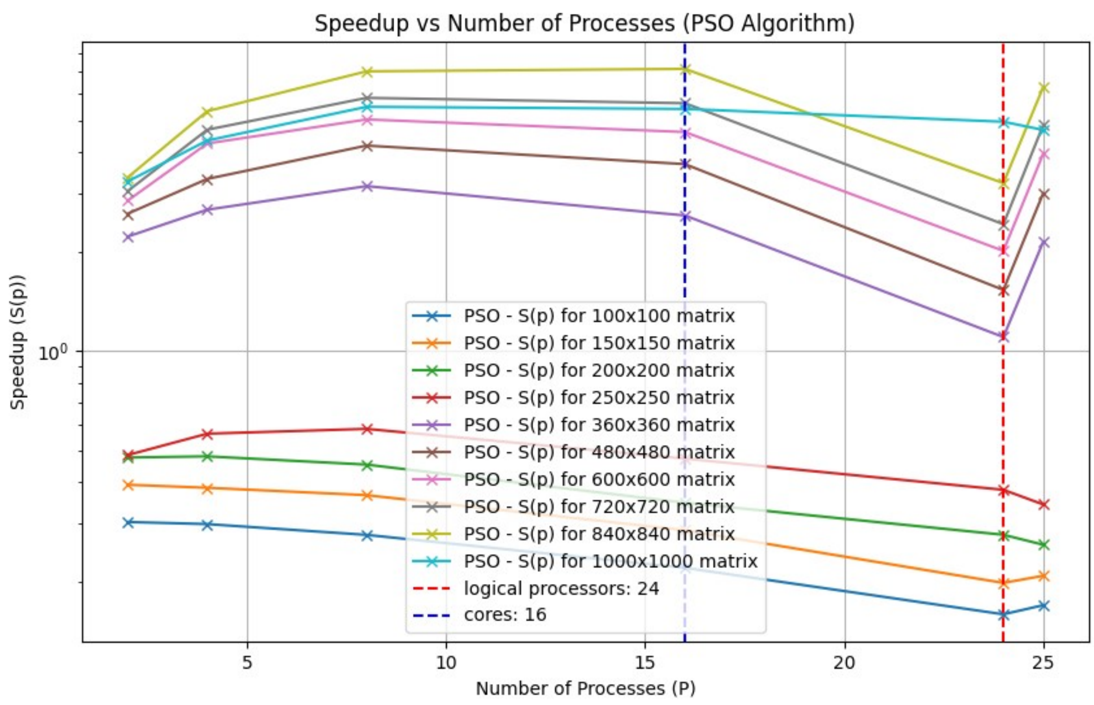

Speedup vs Matrix Size (PSO Algorithm) - Logarithmic Scale
Speedup (S(p))
Matrix Size (N)
10^0
200
400
600
800
1000
PSO - S(p) for 2 processes
PSO - S(p) for 4 processes
PSO - S(p) for 8 processes
PSO - S(p) for 16 processes
PSO - S(p) for 24 processes
PSO - S(p) for 25 processes

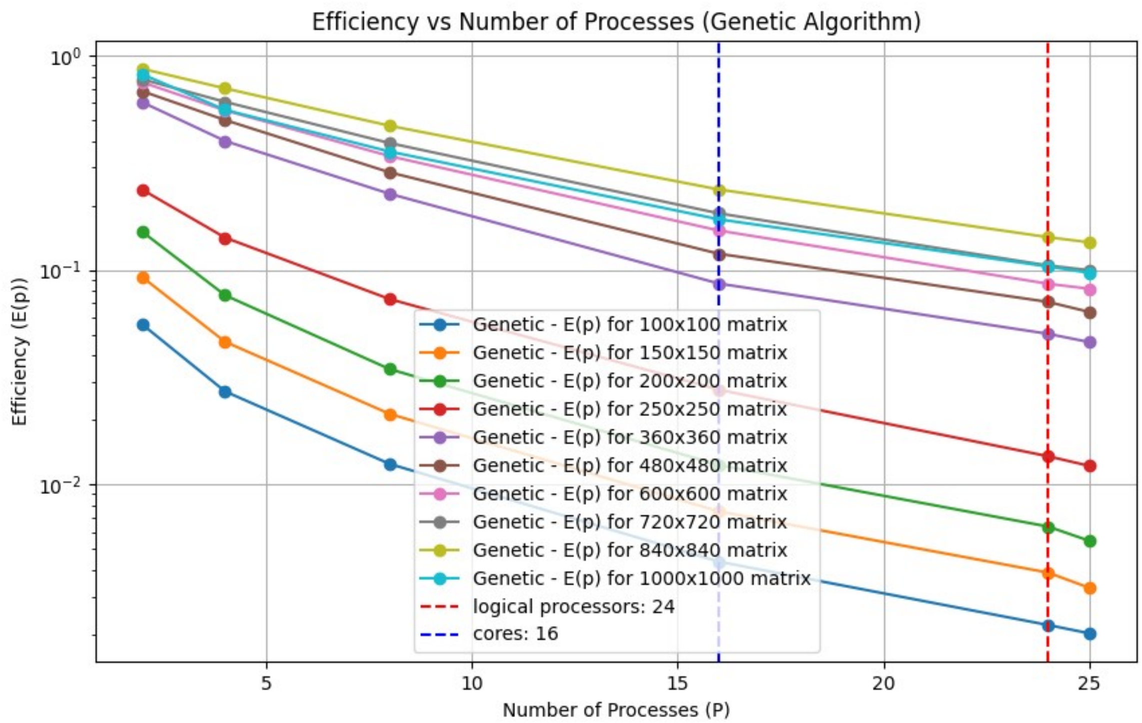

Efficiency vs Matrix Size (PSO Algorithm) - Logarithmic Scale
Efficiency (E(p))
Matrix Size (N)
10^0
10^{-1}
10^{-2}
200
400
600
800
1000
PSO - E(p) for 2 processes
PSO - E(p) for 4 processes
PSO - E(p) for 8 processes
PSO - E(p) for 16 processes
PSO - E(p) for 24 processes
PSO - E(p) for 25 processes

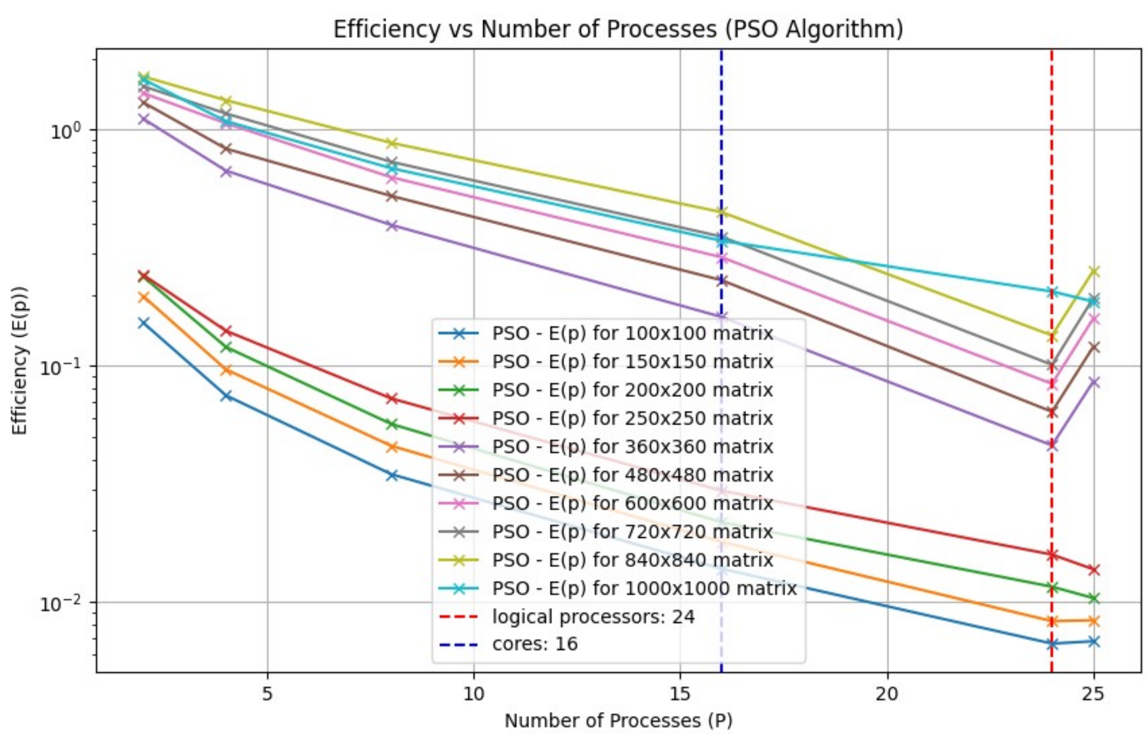

Speedup vs Matrix Size (Genetic Algorithm) - Logarithmic Scale
Speedup (S(p))
Matrix Size (N)
Genetic - S(p) for 2 processes
Genetic - S(p) for 4 processes
Genetic - S(p) for 8 processes
Genetic - S(p) for 16 processes
Genetic - S(p) for 24 processes
Genetic - S(p) for 25 processes


Efficiency vs Matrix Size (Genetic Algorithm) - Logarithmic Scale
Efficiency (E(p))
Matrix Size (N)
Genetic - E(p) for 2 processes
Genetic - E(p) for 4 processes
Genetic - E(p) for 8 processes
Genetic - E(p) for 16 processes
Genetic - E(p) for 24 processes
Genetic - E(p) for 25 processes


Speedup vs Number of Processes (Genetic Algorithm)
Speedup (S(p))
Number of Processes (P)
Genetic - S(p) for 100x100 matrix
Genetic - S(p) for 150x150 matrix
Genetic - S(p) for 200x200 matrix
Genetic - S(p) for 250x250 matrix
Genetic - S(p) for 360x360 matrix
Genetic - S(p) for 480x480 matrix
Genetic - S(p) for 600x600 matrix
Genetic - S(p) for 720x720 matrix
Genetic - S(p) for 840x840 matrix
Genetic - S(p) for 1000x1000 matrix
logical processors: 24
cores: 16


Speedup vs Number of Processes (PSO Algorithm)
Speedup (S(p))
Number of Processes (P)
PSO - S(p) for 100x100 matrix
PSO - S(p) for 150x150 matrix
PSO - S(p) for 200x200 matrix
PSO - S(p) for 250x250 matrix
PSO - S(p) for 360x360 matrix
PSO - S(p) for 480x480 matrix
PSO - S(p) for 600x600 matrix
PSO - S(p) for 720x720 matrix
PSO - S(p) for 840x840 matrix
PSO - S(p) for 1000x1000 matrix
logical processors: 24
cores: 16


Efficiency vs Number of Processes (Genetic Algorithm)
Efficiency (E(p))
Number of Processes (P)
Genetic - E(p) for 100x100 matrix
Genetic - E(p) for 150x150 matrix
Genetic - E(p) for 200x200 matrix
Genetic - E(p) for 250x250 matrix
Genetic - E(p) for 360x360 matrix
Genetic - E(p) for 480x480 matrix
Genetic - E(p) for 600x600 matrix
Genetic - E(p) for 720x720 matrix
Genetic - E(p) for 840x840 matrix
Genetic - E(p) for 1000x1000 matrix
logical processors: 24
cores: 16


Efficiency vs Number of Processes (PSO Algorithm)
Efficiency (E(p))
Number of Processes (P)
PSO - E(p) for 100x100 matrix
PSO - E(p) for 150x150 matrix
PSO - E(p) for 200x200 matrix
PSO - E(p) for 250x250 matrix
PSO - E(p) for 360x360 matrix
PSO - E(p) for 480x480 matrix
PSO - E(p) for 600x600 matrix
PSO - E(p) for 720x720 matrix
PSO - E(p) for 840x840 matrix
PSO - E(p) for 1000x1000 matrix
logical processors: 24
cores: 16