EISTI
École Internationale des Sciences du Traitement de l'Information

www.eisti.fr

SURFACE OPTIMIZATION

HEURISTICS

GUSTAVO FLEURY SOARES
INDURAJ P. RAMAMURTHY

Professor:
PhD. Rachid Chelouah

Cergy / FR
December, 2019

Summary

1 Introduction

Optimization consist in find the best element (maximize or minimize), considering some criterion or boundaries, from some function or set of available alternatives. For some problems when the traditional process of optimization does not give a good result in acceptable time, we use the approach of heuristic optimization. Like explained in [1], "*this is achieved by trading optimality, completeness, accuracy, or precision for speed. In a way, it can be considered a shortcut.*"

These type of optimization techniques start with some initial solutions, and find the best from several moves made.

The subject of this work is try some heuristic optimization for the problem:

"**find the building with the largest rectangular floor area contained in the given area**"

The follow figure exemplifies the problem:

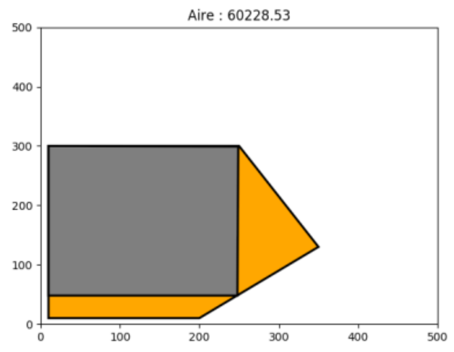
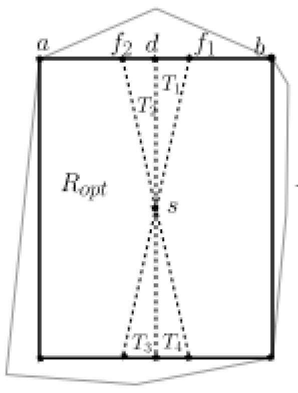
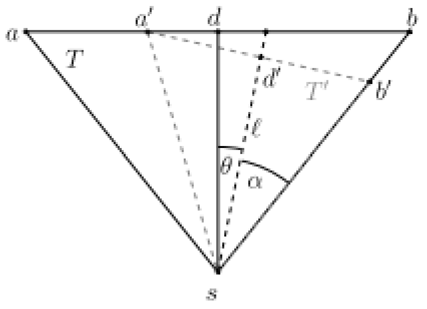


Figure 1. Example of given area and solution.

This work present adaptations of Simulated Annealing and Particle Swarm Optimizations for solve this problem.

2 Problem modeling

We used the proposed solution of the exercise, that is the same presented in [2].

 a f_2 d f_1 b T_1 T_2 R_{opt} s P T_3 T_4	 a a' d b T d' T' b' ℓ θ α s
Figure 2. Largest rectangle in poly	**Figure 3.** Values to characterize the rectangle: Points A, O and Angle.

For initialize the random rectangle, we limited the position of point O and A inside of max and min values of axes x and y. After this we verify with result rectangle are inside of the constrain polygon.

For calculate the area of rectangle, we use the simple logic:

```
def distance(p1,p2):
    return sqrt((p1[0]-p2[0])**2 + (p1[1]-p2[1])**2)

def aire(p1):
    l= length(p1[0],p1[1])
    b= breath(p1[1],p1[2])
    return l*b

def length(a,b):
    return abs(a[1]-b[1])

def breath(b,c):
    return abs(b[0]-c[0])
```

To have a parameter to compare between the solutions, we used the computation time and percentage of total area of the polygon. To calculate the total area, we used the following code:

```
def PolygonArea(corners):
    n = len(corners) # of corners
    area = 0.0
    for i in range(n):
        j = (i + 1) % n
        area += corners[i][0] * corners[j][1]
        area -= corners[j][0] * corners[i][1]
    area = abs(area) / 2.0
    return area
```

With this area, our tests polygons are:

```
( polygone, maxPolArea ) = ( ((10,10),(10,400),(400,400),(400,10)) , 152100)
( polygone, maxPolArea ) = ( ((10,10),(10,300),(250,300),(350,130),(200,10)) ,
81100)
( polygone, maxPolArea ) =( ((50,150),(200,50),(350,150),(350,300),(250,300),(2
00,250),(150,350),(100,250),(100,200)) , 46000)
( polygone, maxPolArea ) =( ((50,50),(50,400),(220,310),(220,170),(330,170),(33
0,480),(450,480),(450,50)) , 167000)
```

3 Simulated Annealing Solution

Simulated Annealing –SA is a metaheuristic probabilistic technique of optimization. The name come from annealing in metallurgy. The simulation is performed using a stochastic sampling method, the Metropolis algorithm.

3.1 Solution

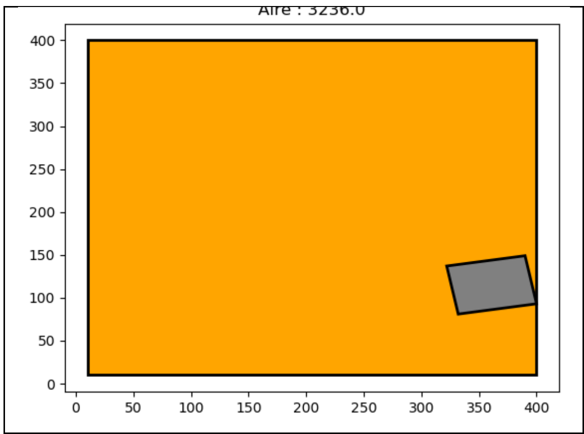
We adapted the algorithm using the following logic:

```
for i in range(IterMax):
    for pointVar in lstVar:
        while (iterStep > 0):
            neighbor = randomNeighbor(rectSol, polygone, pointVar)
            rectSol = metropolis(neighbor,rectSol,T, pointVar)
            iterStep -= 1

        # cooling law enforcement
        t += 1
        T = T*Alpha
        iterStep = Step
```

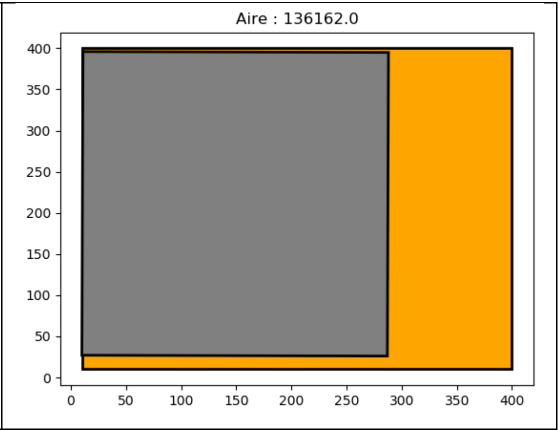
3.2 Results

The tests are done with the following parameters:

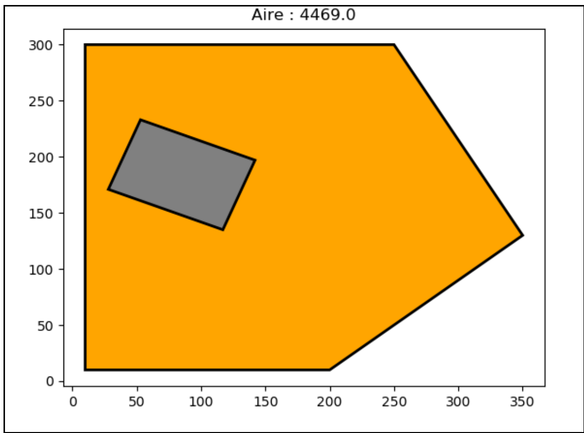
```
## PARAMETERS:
T0 = 10             # initial temperature
Alpha = 0.9         # constant for geometric decay
Step = 7            # number of iterations on a temperature level
IterMax = 15000     # 15000 max number of iterations of the algorithm

sizeNeigh = 0.8       # Size of the Xmax-Xmin to randomize
anglemin = 0.01
anglemax = 89.99
```



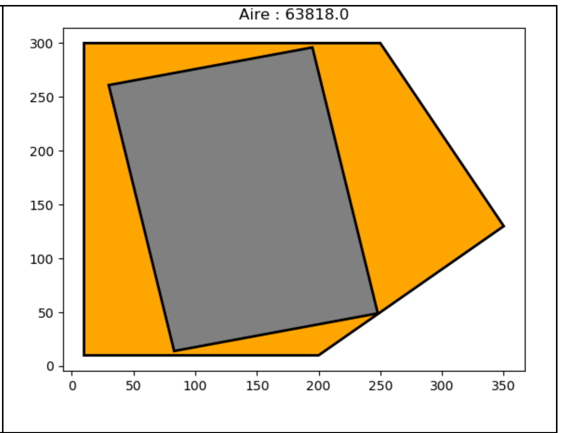
Polygon 1. Initial Random.



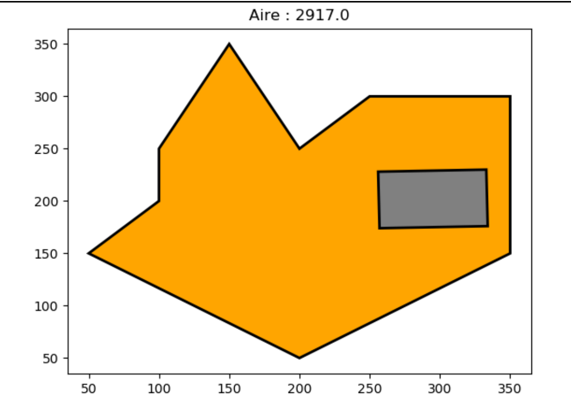
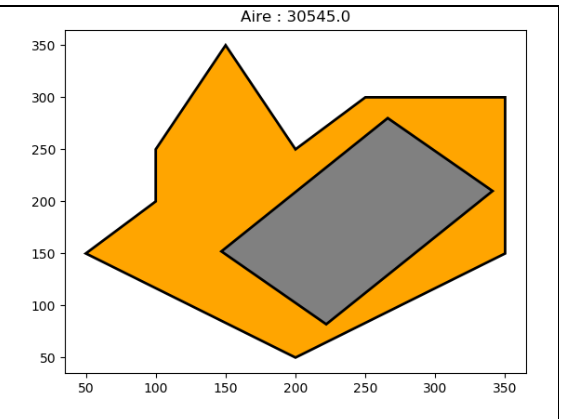
Final

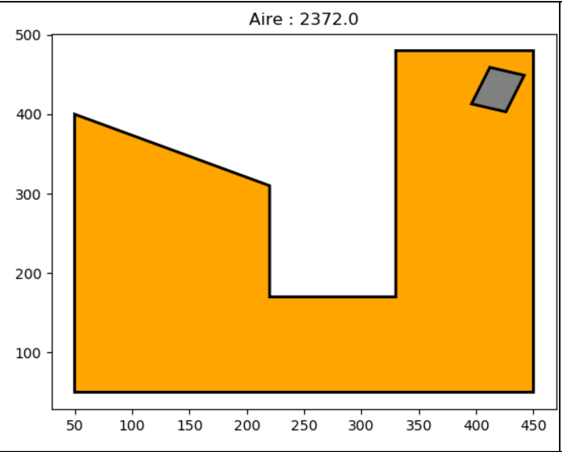
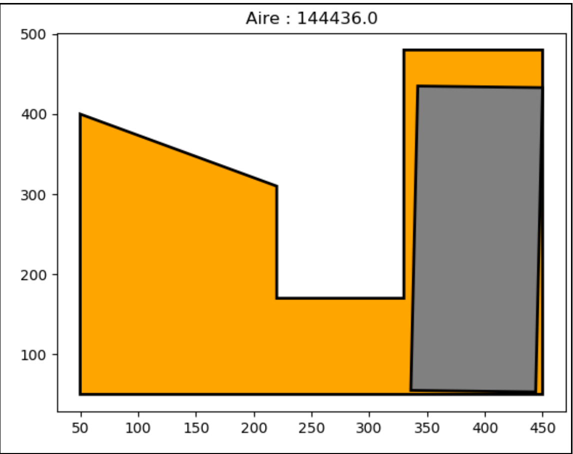


Polygon 2. Initial Random.



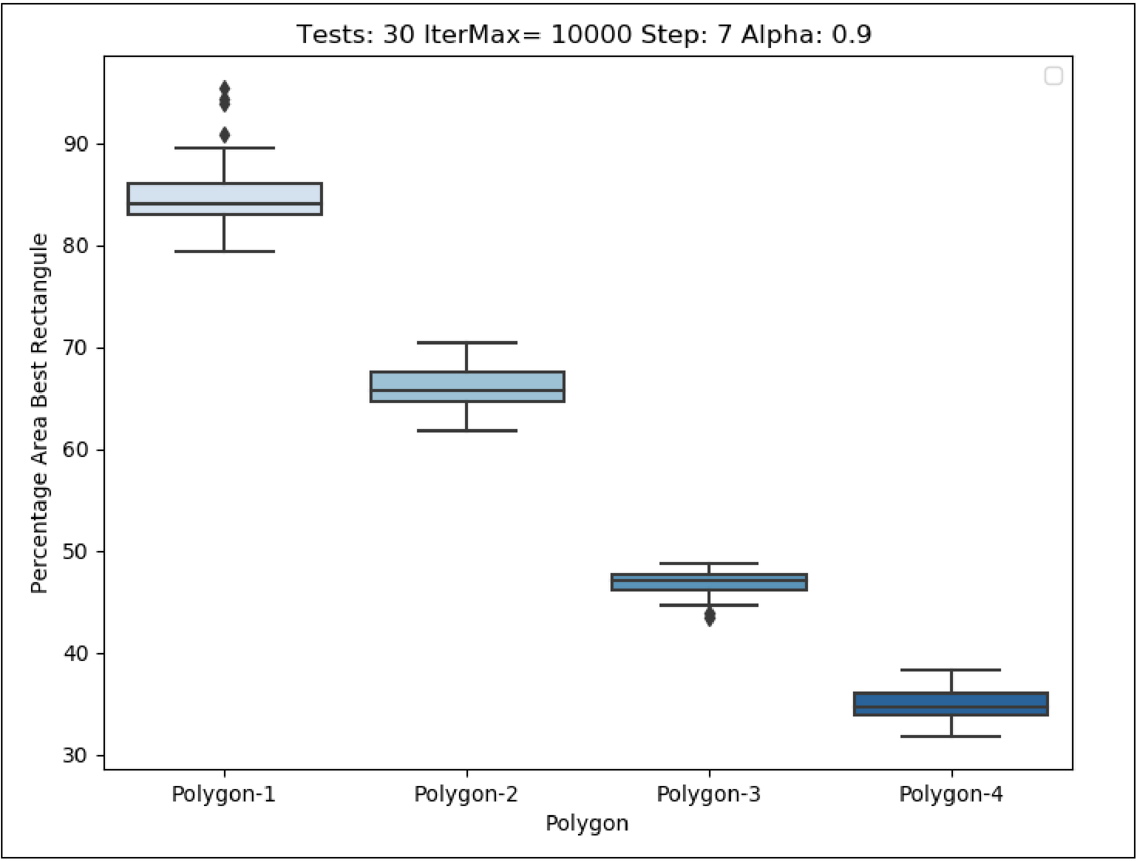
Final

	
Polygon 3. Initial Random.	Final

	
Polygon 4. Initial Random.	Final

3.3 Sample of 30 results and Tuckey Boxes

We ran the code for each polygon 30 times and take the best values. All test used the same initial configurations. The results shows that the algorithm gives similar results (small standard deviation.



4 Particle Swarm Optimization

The Particle Swarm Optimization solves the problem using a certain number of particles that are randomly initialized in the exploration area. These particles move around using a weight value of position and velocity. Each particle uses the information of the best solution and local best.

4.1 Solution

For implement the discrete PSO with the Surface problem, using the auxiliary functions show in topic 2. The PSO main function is show above:

```python
def main_PSO(Nb_Cycles, Nb_Indiv, psi, c1, c2 ):
    global best
    Htemps = []
    Hbest = []
    pos = initPop(Nb_Indiv,polygone)
    best = getBest(pos)
    best_cycle = best
    polygonefig = poly2list(polygone)
    k=0
    z=0
    no_iteration=0
    ltaboo=10
    best_of_best=[0]
    best_of_best[0]=best.copy()
    for z in range(Nb_Cycles):
        pos=[update(e,best_cycle) for e in pos]
        pos=[move(e,psi,c1,c2) for e in pos]
        best_cycle=getBest(pos)
        if best_cycle['bestfit']>best_of_best[0]['bestfit']: #Update Best_of_Best
            best_of_best[0]=best_cycle.copy()
        if(best_cycle['bestfit']>best['bestfit']):
            best=best_cycle
        else:
            no_iteration=0
            rand_pos = initUn(polygone)
            (best_cycle['bestpos'], best_cycle['fit']) = metropolis(rand_pos['pos']
, rand_pos['fit'], best['bestpos'], best['bestfit'],0.1) #10000000
        if z % 10 == 0:
            dessine(polygonefig, poly2list(post2rect(best_of_best[0]["bestpos"])),
z)
            Htemps.append(z)
            Hbest.append(best_of_best[0]['bestfit'])
```

4.2 Parameters Tuning

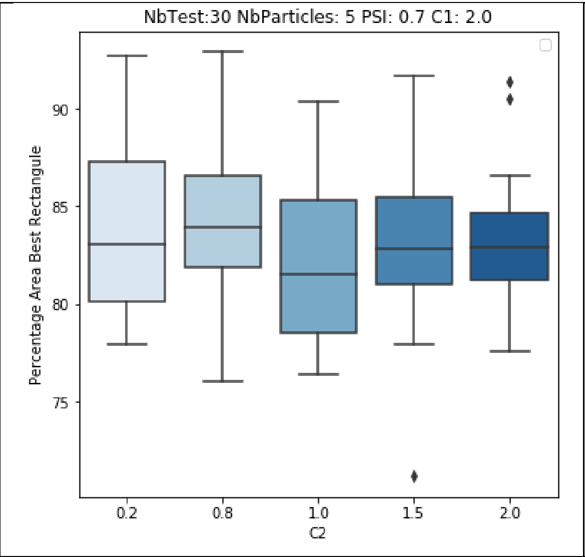
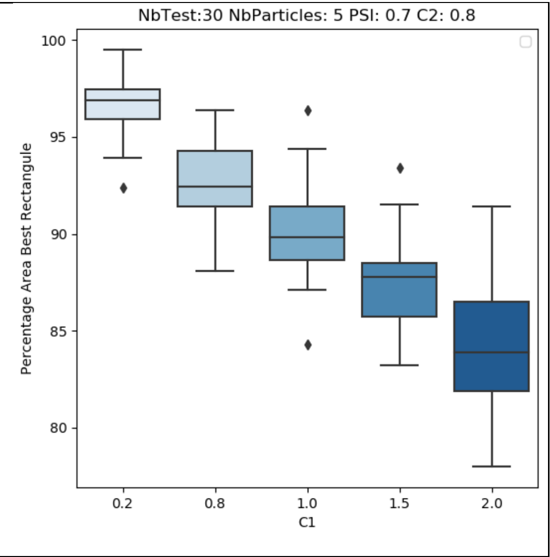
For select the best parameters for this problem, we tested each parameter for 30 times, and select the best value for each parameter. The logic used is shown in code above. The file with this code is "*surface_PSO - Test Parameters.py*".

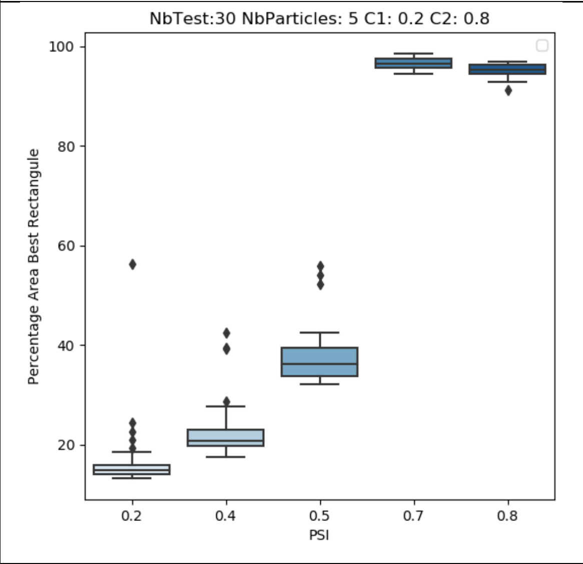
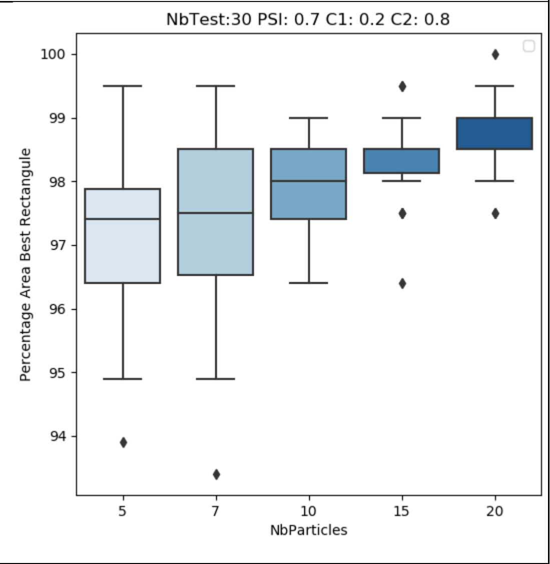
```
BoxPlotData = []
for Nb_Cycles in Nb_Cycles_LST:
    for Nb_Indiv in Nb_Indiv_LST:
        for psi in psi_LST:
            for c1 in c1_LST:
                for c2 in c2_LST:
                    HAvgBest = []
                    for i in range(numTests):
                        start_time = time.time()
                        HAvgBest.append( main_PSO(Nb_Cycles, Nb_Indiv, psi, c1, c2))
                        HComputationTime.append( time.time() - start_time )
                    BoxPlotData.append( HAvgBest )
```

The values tested for each parameter are shown above.

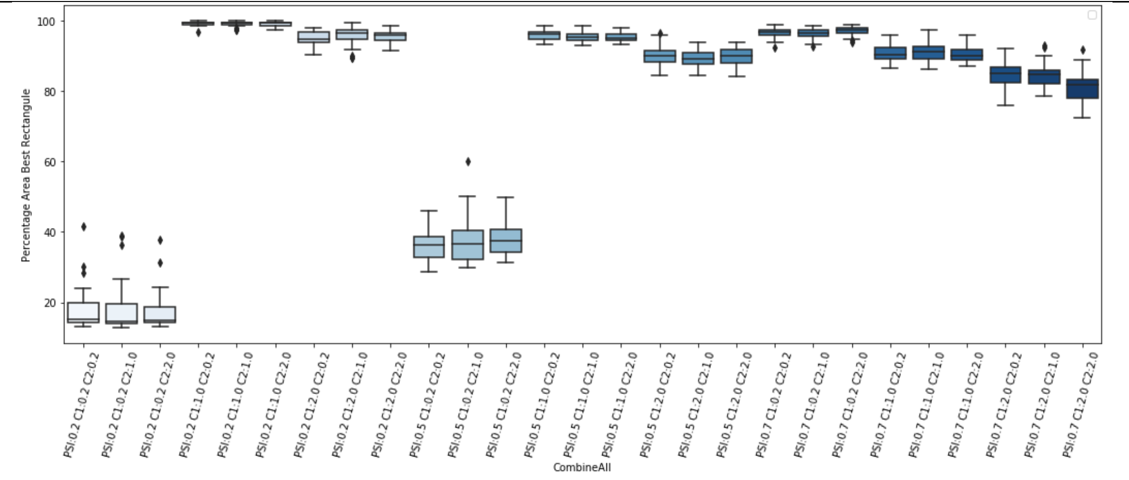
```
## PARAMETERS:
Nb_Cycles_LST = [10000]  # [500, 1000, 1500, 5000, 10000, 20000]
Nb_Indiv_LST = [5]       # [5, 7, 10, 15, 20]
psi_LST = [0.7]          # [0.2, 0.4, 0.5, 0.7, 0.8]
c1_LST = [2.0]           # [0.2, 0.8, 1.0, 1.5, 2.0]
c2_LST = [0.2]           # [0.2, 0.8, 1.0, 1.5, 2.0]
```

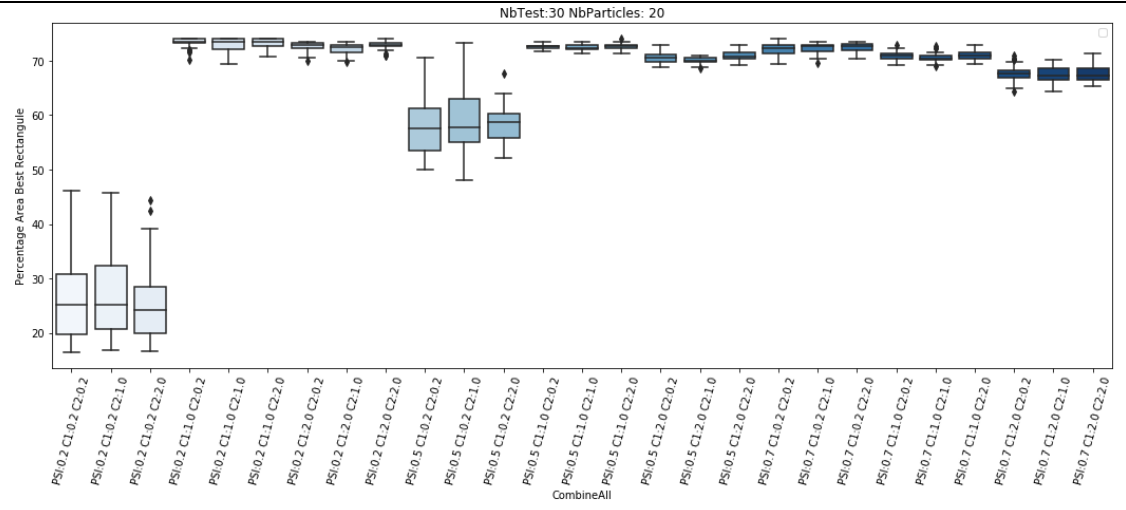
The following table show the result boxplot for each parameter.

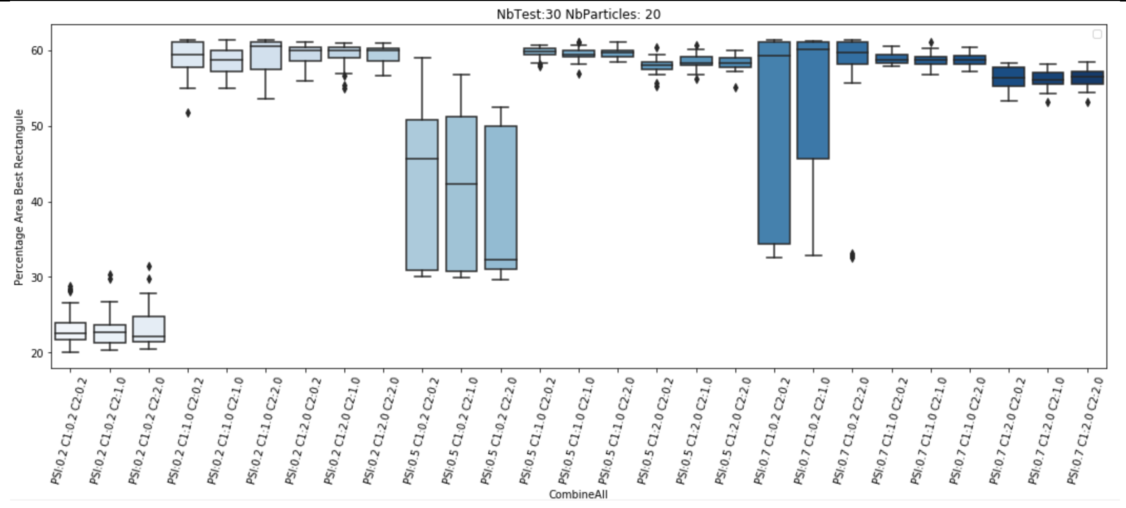
C2 – Best Value: 0.8	C1 – Best Value: 0.2
	
PSI – Best Value: 0.7	Number of Particles – Best Value: 20

With more particles we got better results, but the time of response get exponentially bigger too. The next graph shows the test changing PSI, C1 and C2.

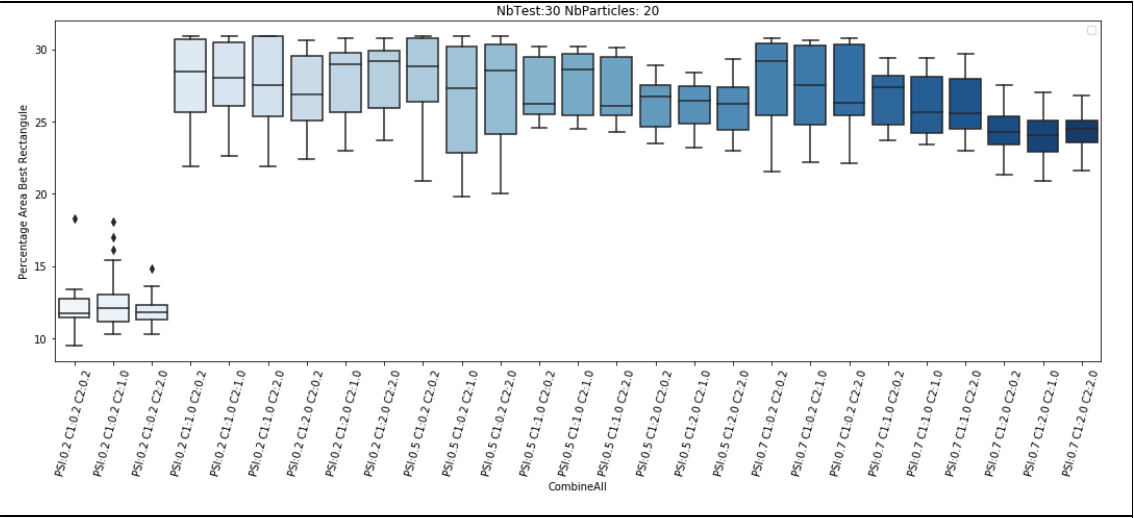
Using all the combination of PSI [0.2, 0.5, 0.7], C1[0.2, 1.0, 2.0] and C2[0.2, 1.0, 2.0].

Polygon 1. Best values at PSI: 0.2 C1: 1.0 e C2: 1.0



Polygon 2. Best values at PSI: 0.2 C1: 1.0 e C2: 2.0



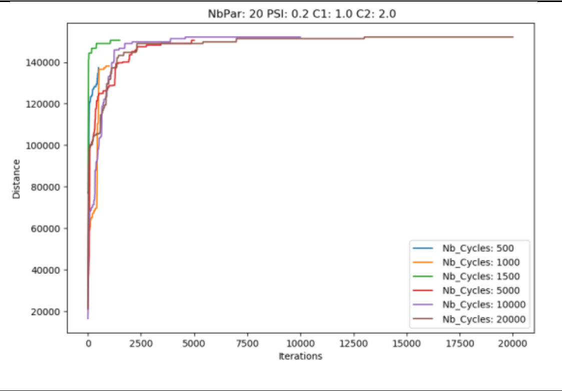
Polygon 3. Best values at PSI: 0.2 C1: 1.0 and C2: 2.0



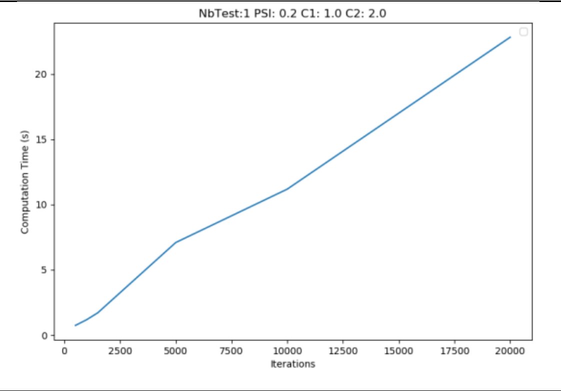
Polygon 4. Best values at PSI: 0.2 C1: 1.0 and C2: 2.0

The best values overall could be simplified to the PSI: 0.2 C1: 1.0 and C2: 2.0.

About the number of iterations, we can see that after 10.000, there are acceptable result and few small increments, but we got a big increase in the computation time. We used Nb_Cycles equal 10.000.



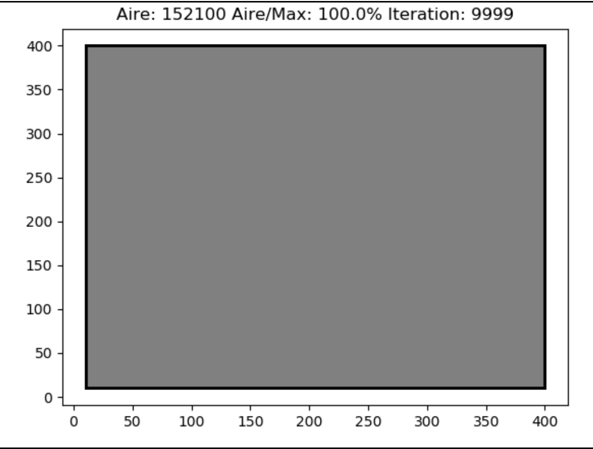
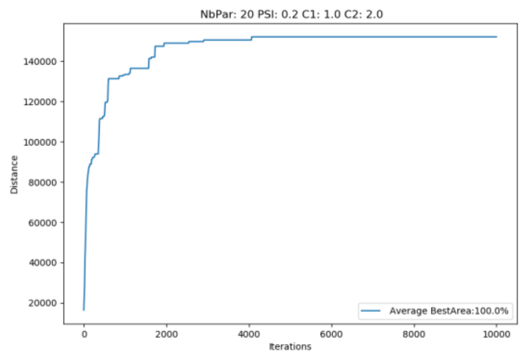
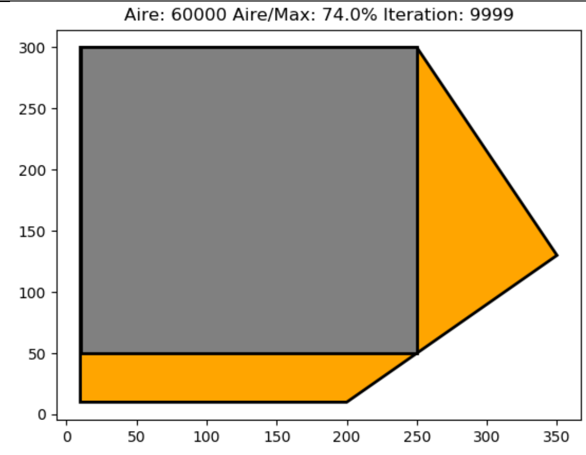
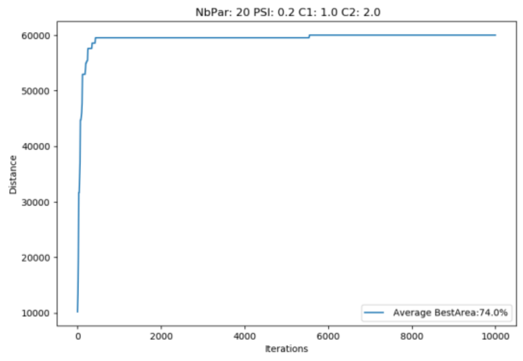
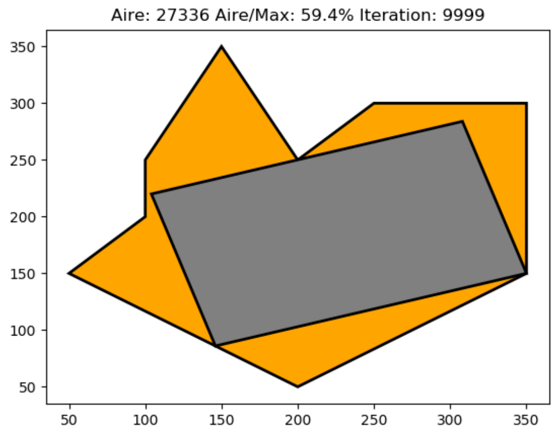
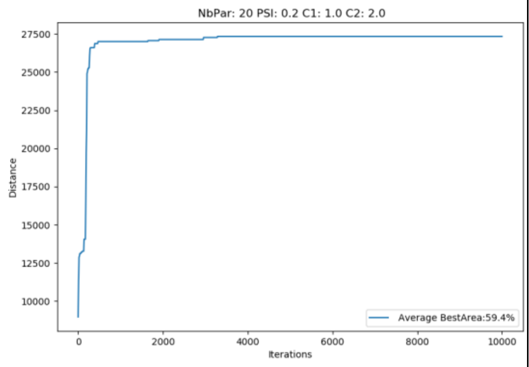
Increments per Number of Cycles.

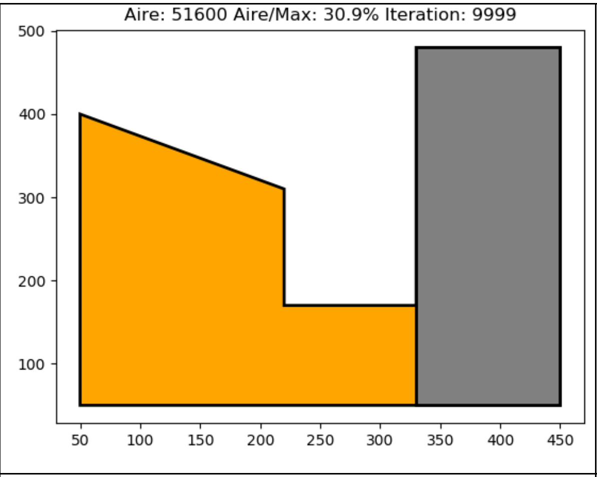
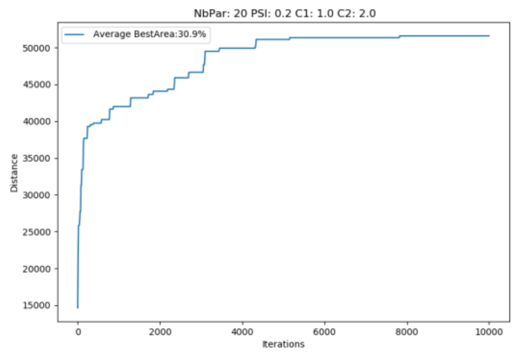


Computation time (seconds) per Number of Cycles.

4.3 Results

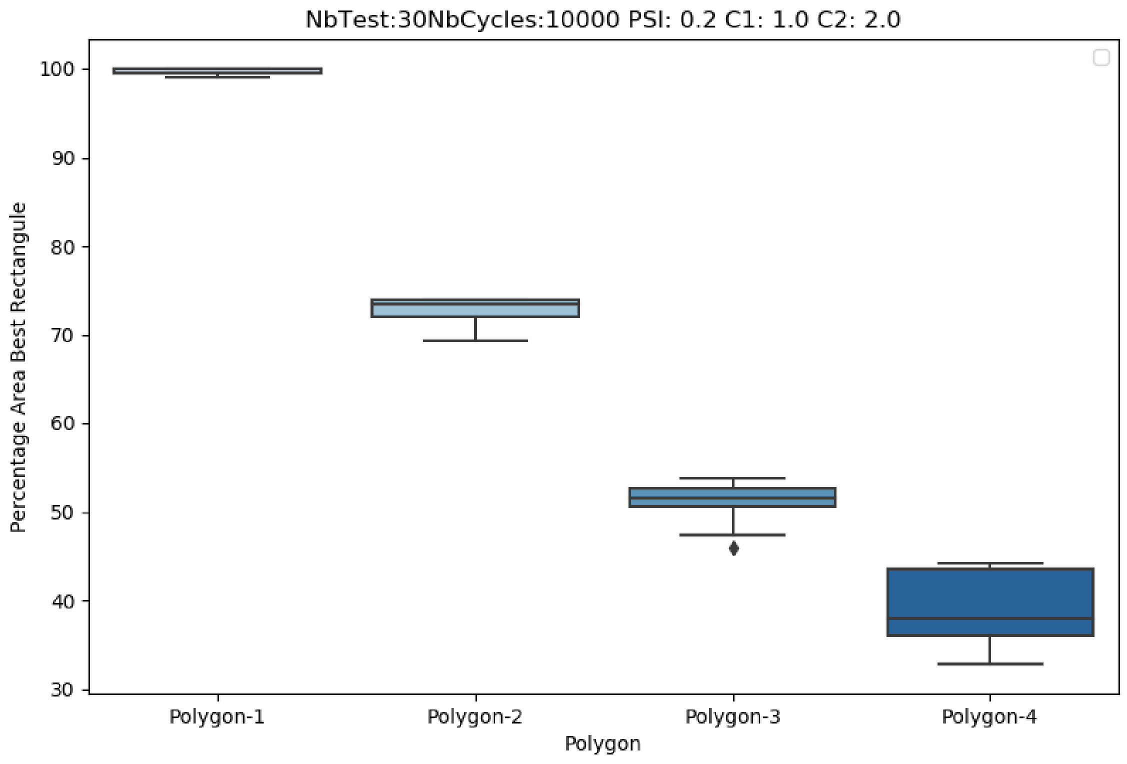
Using the parameters found in previous topic, we tested the solution for each polygon. We verify that with using the tuned parameters, the algorithm converges faster and to a best final area. This images show an example of result for each polygon.

	
Polygon 1. 100% of Max Area	Improvements Polygon 1.
	
Polygon 2. 74% of Max Area	Improvements Polygon 2.
	
Polygon 3. 59.4% of Max Area	Improvements Polygon 3.

	
Polygon 4. 26.1% of Max Area	Improvements Polygon 4.

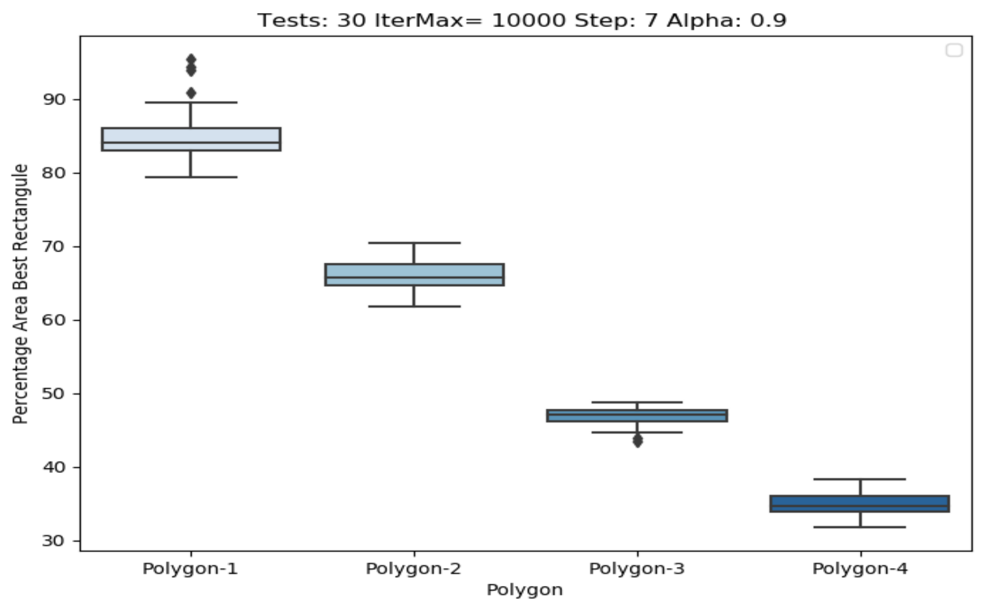
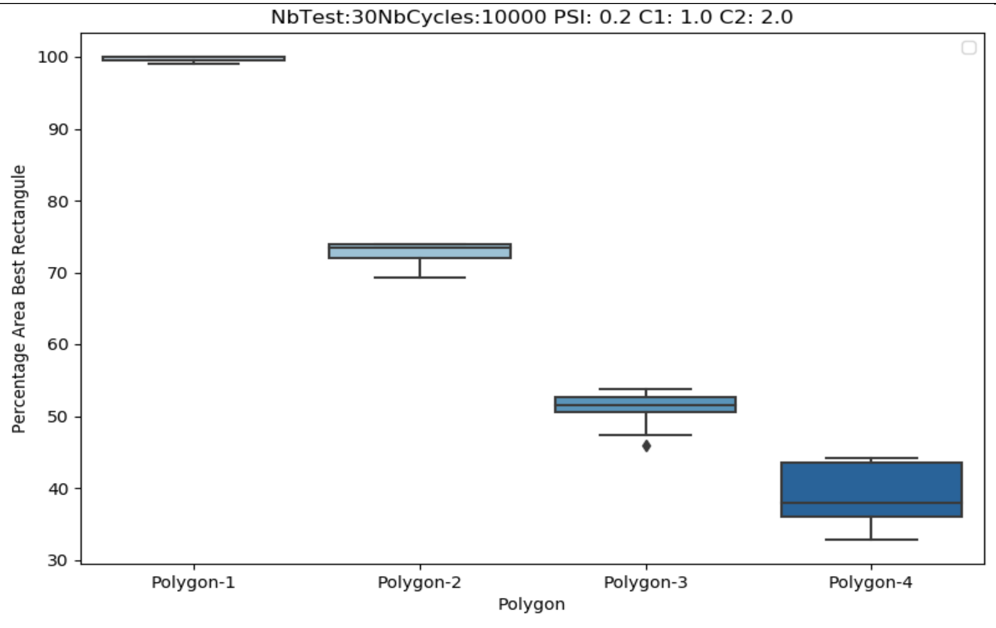
4.4 Sample of 30 results and Tuckey Boxes

We tested the algorithm for each polygon for 30 times and show the result in the following Boxplot. We can observe that for the polygon 1, the variance of result is very small. For the polygon 4, sometimes the algorithm could stay in other area, so it gives a bigger variation.



5 Algorithms comparison

For compare the algorithms we tested each polygon 30 times and used 10000 iterations for both. The figures above show the results, where we can compare the quality. Our PSO solution gives the best results for all polygons, but SA is more stable, with less standard deviation.

Simulating Annealing Algorithm Results	
PSO Algorithm Results	

6 Bibliographic References

[1] MIT, Heuristic Design and Optimization.

[2] C. Knauer, L. Schlipf, J. Schimidt and T. Hans, Largest inscribed rectangles in convex polygons, University Bayreuth, 2011.