

KU LEUVEN

GENETIC ALGORITHMS

TSP

Author:

Alejandro RODRÍGUEZ SALAMANCA: r0650814@student.kuleuven.be
Fernando COLLADO EGEA: r0650586@student.kuleuven.be

January 10, 2017

Contents

1 Implementation	2
1.1 Representation	2
1.2 Crossover	2
1.3 Mutation	3
1.4 Fitness Function	3
2 Experiments	3
2.1 General tests	4
2.1.1 Original code	4
2.1.2 Modified code	5
2.2 Specific tests	6
2.2.1 Original code	6
2.2.2 Improved code	9
3 Optional part	12
4 Appendix	16
4.1 tsp_ImprovePopulation.m	16
4.2 run_ga.m	17
4.3 insertion.m	19
4.4 order_crossover.m	20
4.5 order_low_level.m	21
4.6 tspgui.m	22
4.7 tspfun.m	22
4.8 mutateTSP.m	23
4.9 run_ga_test.m	24
4.10 tspgui_test.m	26
4.11 test.m	32
4.12 select_rr.m	39
4.13 Graph	41
4.13.1 Original-General-Individuals	41
4.13.2 Original-General-Generations	42
4.13.3 Original-General-Elitism	42
4.13.4 Original-General-Crossover and mutation	43
4.13.5 Modified-General-Individuals	43
4.13.6 Modified-General-Generations	44
4.13.7 Modified-General-Elitism	44
4.13.8 Modified-General-Crossover and mutation	45

1 Implementation

1.1 Representation

The original code employed adjacency representation. In adjacency representation a tour is represented as a list of n cities where city j is listed in position i if and only if the tour leads from city i to city j . Thus, the list:

$$(7 \ 6 \ 8 \ 5 \ 3 \ 4 \ 2 \ 1)$$

represents the tour:

$$3-8-1-7-2-6-4-5$$

¹

In our implementation, we have decided to use path representation. Path representation is the most natural way of representing a tour. This can be easily seen with the following example. The list:

$$(1 \ 2 \ 7 \ 5 \ 6 \ 3 \ 4)$$

represents the path

$$1-2-7-5-6-3-4$$

It was also the simplest representation possible for this problem, and it was easy to implement, as the tour was first encoded in path representation, and then translated to adjacency representation using the function `path2adj`. Finally, it was translated again to path representation to be used in the plots with the function `adj2path`.

1.2 Crossover

The new representation required a new crossover operator. The available options were:

- Partially Matched Crossover (PMX)
- Order Crossover (OX)
- Cyclic Crossover (CX)
- Edge Recombination Crossover (ERX)

The one selected is Order Crossover. This crossover exploits a property of the path representation, that the order of the cities (not their positions) are important. It constructs an offspring by choosing a subtour of one parent and preserving the relative order of cities of the other parent. Let's consider the following tours:

$$\begin{aligned} &(1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8) \\ &(2 \ 4 \ 6 \ 8 \ 7 \ 5 \ 3 \ 1) \end{aligned}$$

¹This explanation can be found in the slides about Traveling Salesman Problem

If we choose the cut point between the second and the third city, and the second cut point between the fifth and the sixth, we have:

$$\begin{aligned} & (1 \ 2 — 3 \ 4 \ 5 — 6 \ 7 \ 8) \\ & (2 \ 4 — 6 \ 8 \ 7 — 5 \ 3 \ 1) \end{aligned}$$

And the offspring is created copying the segments between the cut points, and then, starting from the second cut points of one parent, the rest of the cities are copied in the order in which they appear in the other parent, giving:

$$\begin{aligned} & (8 \ 7 — 3 \ 4 \ 5 — 1 \ 2 \ 6) \\ & (4 \ 5 — 6 \ 8 \ 7 — 1 \ 2 \ 3) \end{aligned}$$

1.3 Mutation

We were also asked to implement a new mutation operator. For this task, multiple and different options where available, such as Exchange Mutation, Scramble Mutation, Displacement Mutation or Insertion Mutation. As the crossover operator chosen exploits the order property aforementioned, we decided that the mutation should break this order in some way to avoid reaching a local optima due to the lack of diversity in the order.

The mutation operator selected for this problem is insertion mutation due to its simplicity and effectiveness. This mutation works in the following way. Imagine that we have the path:

$$0 \ 1 \ \mathbf{2} \ 3 \ 4 \ 5 \ 6 \ 7$$

Take the 2 out of the sequence,

$$0 \ 1 \ 3 \ 4 \ 5 \ 6 \ 7$$

and reinsert the 2 at a randomly chosen position:

$$0 \ 1 \ 3 \ 4 \ 5 \ \mathbf{2} \ 6 \ 7$$

1.4 Fitness Function

The fitness function has been changed as the original one worked with adjacency representation. The simplest way to adapt the old function to the new representation was using `path2adj`, converting the path to adjacency representation, and then computing the fitness in the same way as it was computed before.

2 Experiments

In order to properly study and select the different configurations, it is necessary to analyze the behaviour of the algorithm, when we modify one parameter at a time. An automated test was developed for this purpose, provided in the annex, with which we run a set of tests,

changing the number of individuals, the number of generations, the percentage of elitism, and the balance of percentage of mutation and crossover. Each test, in each city is run a total of 5 times, and the average is what is taken into account.

After that, a set of tests, created from the results obtained in the general set, is made, by selecting specific values. Both the general, and the specific tests will serve as a medium to compare both representations, in order to look for differences (or lack thereof)

2.1 General tests

For the general tests, the default values are:

- Number of individuals - 50
- Number of generations - 50
- Elitism - 0.05
- Crossover - 0.95
- Mutation - 0.05
- Stop percentage condition - 0.95
- Detection of loops on

And, the different values for each modified parameter are:

- Number of individuals - [50,100,150,200,500,750,1000]
- Number of individuals - [50,100,150,200,500,750,1000]
- Elitism percentage - [0,0.05,0.1,0.2,0.5,0.75,1]
- Crossover—Mutation balance - [1—0,0.95—0.05,0.9—0.1,0.75—0.25,0.5—0.5,0.25—0.75,0—1]

2.1.1 Original code

Due to space limitations, we will limit ourselves to only discuss the results of the general tests, while referring to the correspondent graph in the appendix.

When it comes to number of individuals, as expected, a relatively low number of individuals does not provide adequate results, as can be observed by looking at the start of the graph (3.13.1). Almost all datasets start in global maximum, and only a couple in a local max. However, the majority of them have one of their lowest point when the number of individuals is 200 (except for the highest dataset, which has its lowest point at 750) and from that point, it stays constant, or even raises, hence it can be said that any number of individuals higher than 200 would not be beneficial, and actually be just cumbersome when it comes to computational cost.

Regarding the number of generations (3.13.2), once again, it is to be expected that a low quantity of generations will not yield good results. But that is not the only thing the number of individuals and generations have in common, since it seems like 200 is one of the best options for the number of generations. There are some differences, for instance, at lower quantities of generations, there is more fluctuation, and more datasets are positive towards higher number of generations. In the specific tests this will be further studied, whether 200, or a higher number is better, and whether the higher associated computational cost is worth.

When it comes to the percentage of elitism (3.13.3), the results are clearer. With no exceptions, the lowest value for every single one of the datasets is between 0.05 and 0.1, any higher, or any lower, and the distance skyrockets, having the highest distances values at elitism = 1.

This phenomena has an easy explanation, the higher the elitism, the more likely it is that the algorithm will stay at a local maxima.

For crossover and mutation (3.13.4), the test reflects perfectly the balance between exploitation and exploration, the overall result shows that the best performance comes when mutation has a value of 0.5, and crossover 0.5.

Any value of mutation higher than 0.5, and the results start to worsen, because there is too much exploitation, and too little exploration. Any value of mutation lower than 0.5, and the results, most of the cases, are far worse. This leads to the conclusion that 0.5 is the candidate for the specific tests, although the nature of the result makes it necessary to test other values, since, **we think these results are a bit odd, hence we will further study in the specific tests, order to make a conclusion**

2.1.2 Modified code

As explained in the implementation section, the representation we implemented is path representation. Again, we will only discuss the results, the graphs can be found in the appendix

Similarly to adjacency representation, the result for the test (3.13.5) of increasing the number of individuals has a generally located minimum local at 200, although for some cases, the distance becomes constant at 100. The only noticeable difference is that the values for the distances when the number of individuals is very high (750,100) does not increase, rather it seems to keep ever so slightly decreasing.

With regards with number of generations(3.13.6), once again, as expected, there is a number of individuals from which the change in the result is almost insignificant. That point, as can be observed, is 200 individuals, the same as the other representation. One difference is that the values obtained from 750 individuals forward is actually worse in some cases, if not the same, while with the other representation there were some cases in which it improved.

For the elitism(3.13.7), there is no doubt that 0.05 is the best percentage of elitism that can be chosen with this representation. The results are somewhat similar to the previous representation, but the slope at the latest values (0.5 forward) is not so steep

Lately, the result of the crossover and mutation (3.13.8) test is daunting. It was repeated, in case it was somehow erroneous, but the same graph was obtained. With our representa-

tion, there seems to be no effect whatsoever on the exploitation vs exploration dilemma. It does not matter, apparently, the percentage of mutation or crossover

2.2 Specific tests

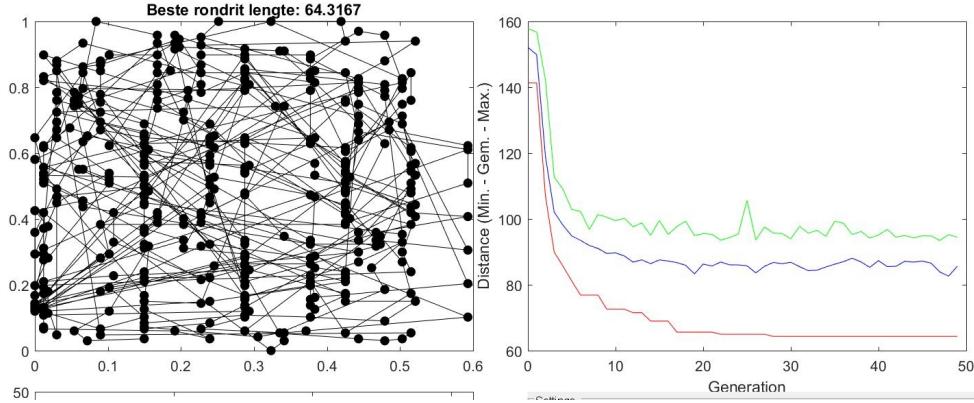
The specific tests are separated in 2 phases.

First phase, study the impact of the best parameter values seen in the general tests, for each representation individually. The second phase corresponds to the comparison between the representation results, first with the base case, and then with the best result obtained. All the tests have been made with the benchmark dataset called bcl380.tsp.

2.2.1 Original code

The base values are the same as with the general case. [Number of individuals, generations, percentage of elitism, of crossover and mutation] = [50,50,5%,95%, 5%]

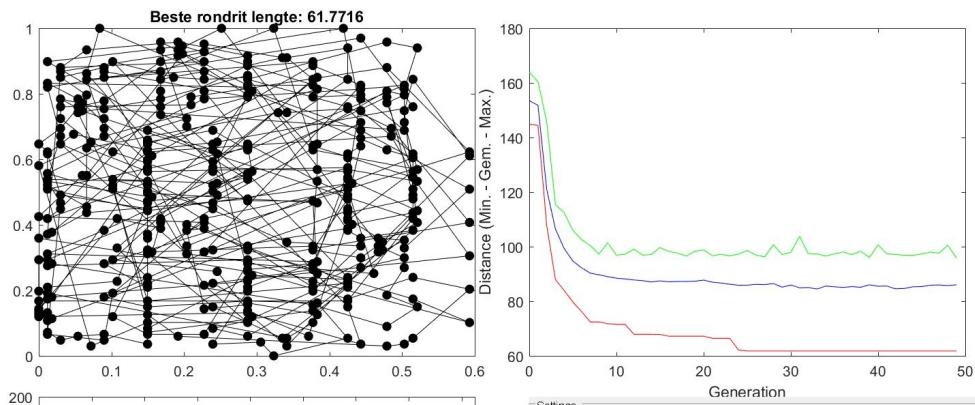
The result obtained is



The test was executed in 12.66 sec

200 was the number of individuals that was deemed to be appropriate. The test, keeping all other values, but changing just the number of individuals is:

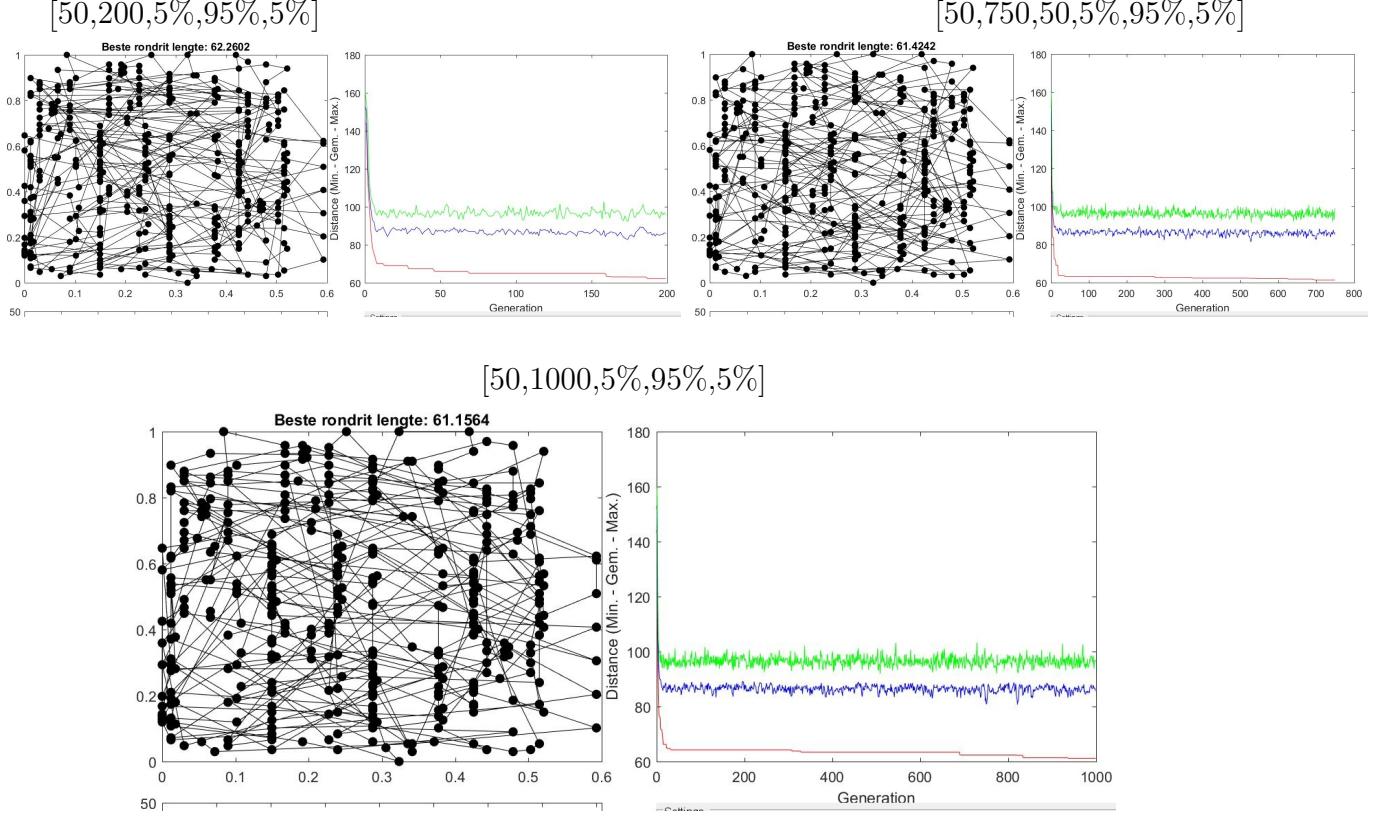
[Nind, gens, elitism, crossover, mutation] =[200,50,5%,95%,5%]



We can see that the minimum distance has decreased, from 64 to 61, not a big decrease, but significant enough. Increasing the number of individuals to 200 is then to be considered a good measure. Timewise, the test was executed in 31.61 seconds. More than twice the time

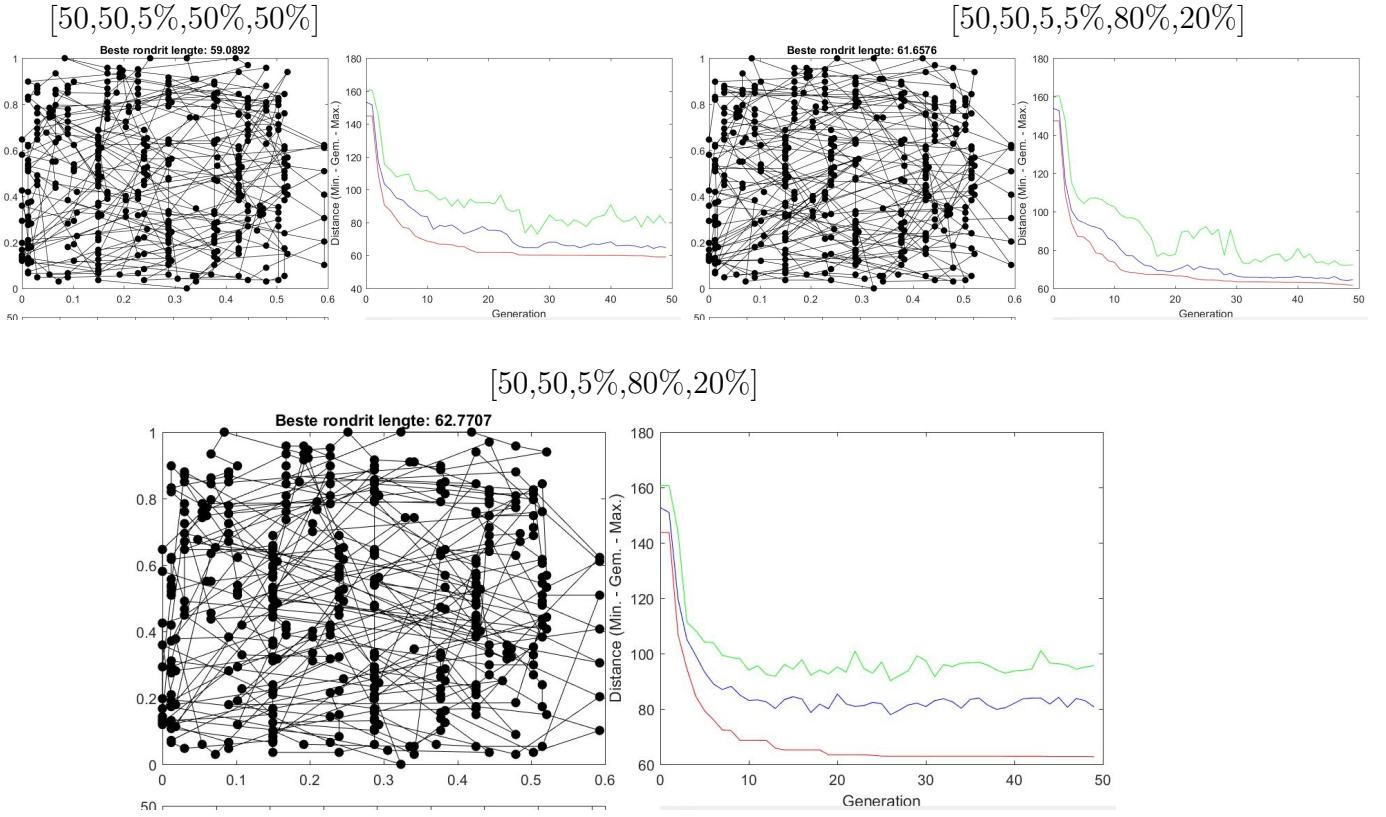
for the first, which, given the increase in the number of individuals, is not out of the expected.

For the next test, this time the number of generations has been increased. As previously stated, the best number was 200, but a higher number was also suitable for some cases. We tried, respectively, with 200, 750, and 1000 generations, thus having



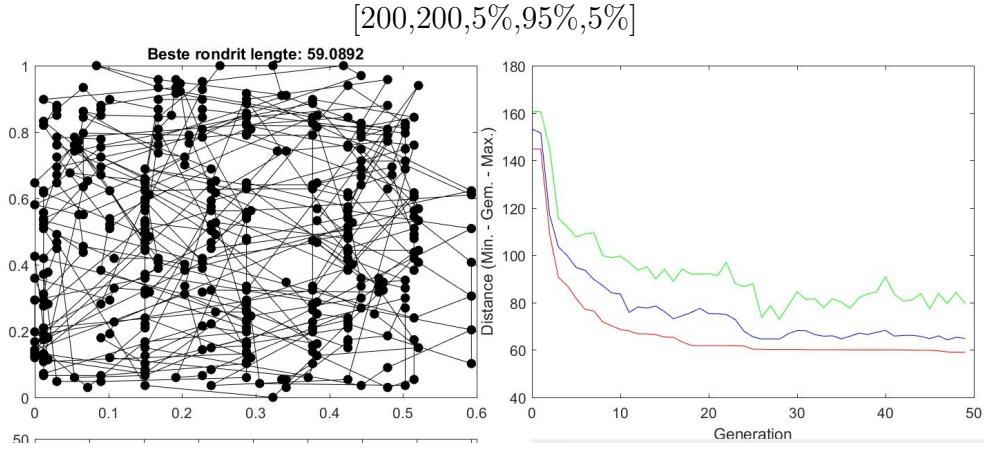
We can observe that the result is slightly better, with each increment of generations, but the cost is excessive, when it comes to time. The times for the tests are 46.01, 171.67, and 233.85 seconds, which means, 4, 13.5 and 18.5 times more than the base case.

As the elitism parameter was abundantly clear that was to be kept at 0.05 (or up to 0.1 at most), and it is already the value for the base case, there is no specific test for the elitism. There is, however, for the percentage of crossover and mutation. Since the result was a bit unexpected, we did not only try with the apparent best result (0.5 percentage of each), but with cases of 0.2 crossover — 0.8 mutation, and viceversa. The results for the tests are

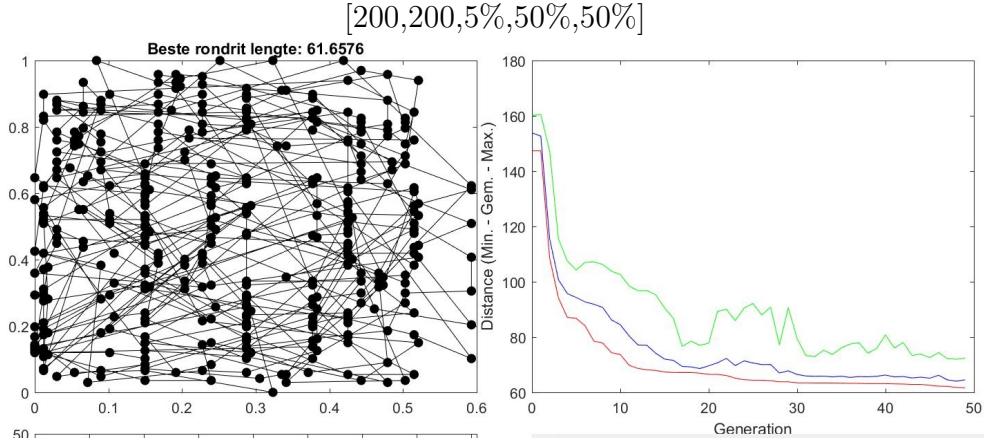


The test confirms the results obtained in the general test. The best ratio of crossover—mutation is 50%—50%, since, out of the 3 configurations, it has the best result, while all of them are over the base case. Although not as relevant in this case, the times of the tests are equal or even lower than the original, 11.3, 9.71, 12.92 seconds respectively.

Finally, the combination of the good results is what made them get outstanding results. In order to avoid bias, the first combination is just modifying the number of individuals, and the number of generations



We can see an improvement, even slightly better than those obtained with the increase in number of individuals or generations alone. The number of generations was chosen to be 200 because we deemed it good enough, and while costly, not as costly as higher quantities.

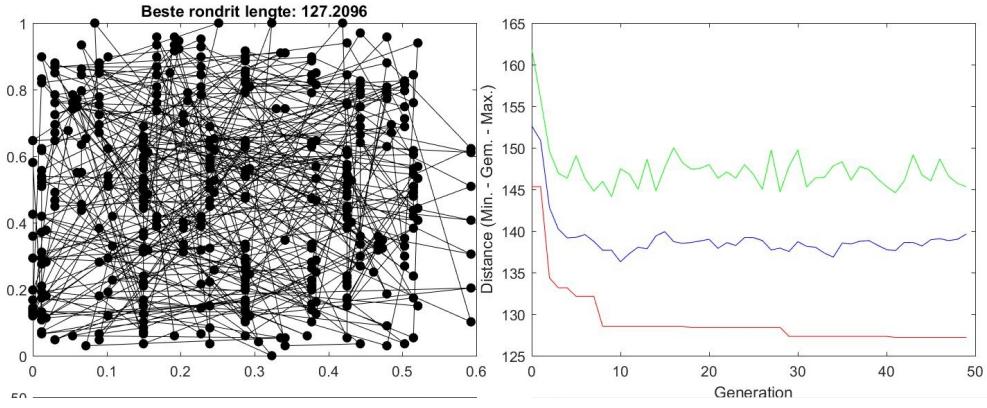


And the final result, when combining every single improvement, is the best of all. The decrease of distance is 30.8%, which could be said to be a great decrease. The time spent on this test was 76.74 seconds, around 6.1 times more than the original, although high, we consider that it is not sufficiently high as to consider it bad.

2.2.2 Improved code

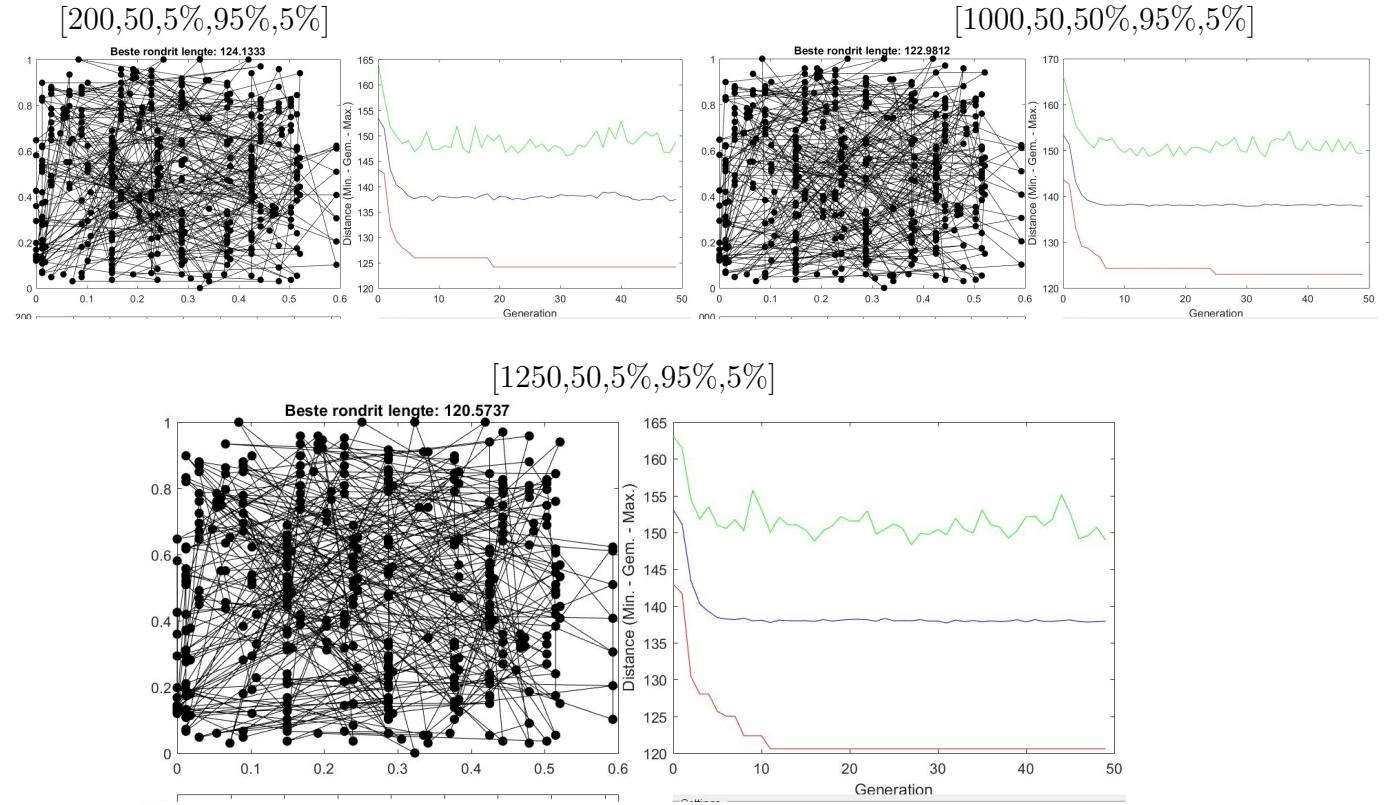
The same structure as the previous representation will be followed. One parameter at a time will be tested, stating the change, after performing a base test

[50,50,5%,95%,5%]



The first noticeable thing that we can observe here is that the distance is far higher than the adjacency representation, although looking at the time, it is better, since it takes only 6.88 seconds.

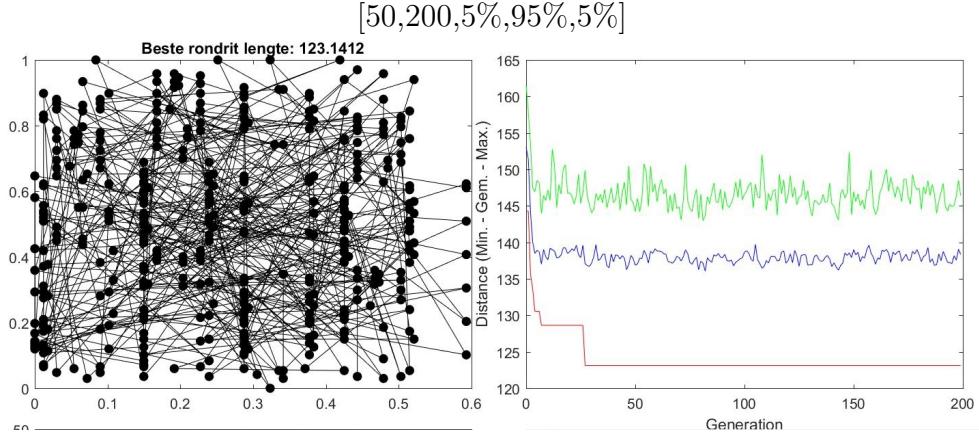
We have the same situation that we had with the number of generations in the previous representation, but this time with the number of individuals. While 200 was the amount with the generally lower distances, higher numbers also performed somewhat ok. Thus we tested for 200, 1000, and 1250.



Although the change is not big, there is improvement, with each increase of individuals. And, what's more, as it is individuals, and not generations that we are increasing now, there is higher cost, but not as high. The times for the tests are 9.08, 29.44, and 37.99 seconds,

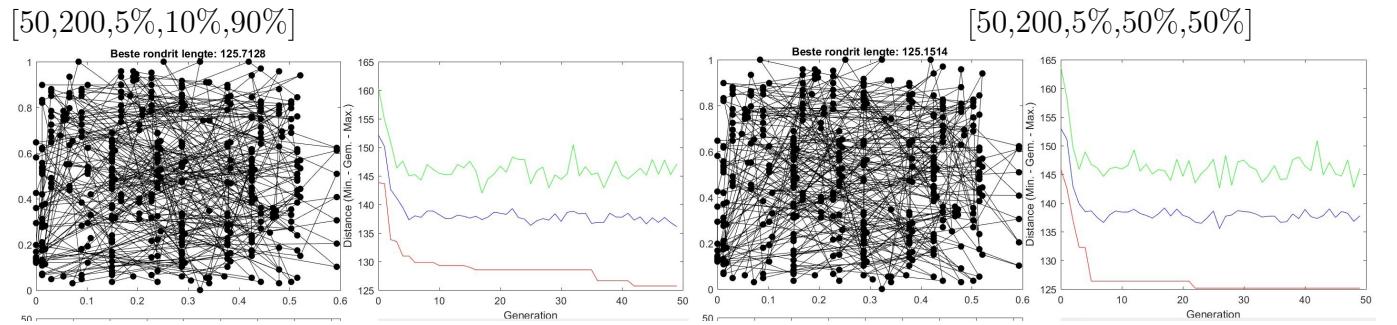
being the last not even 6 times higher. As there is the increase is the highest in the last test (1250 individuals), and the time is not humongously high, that value will be the one to be tested in the combined test.

The general test for the number of generations revealed that the best quantity was 200.



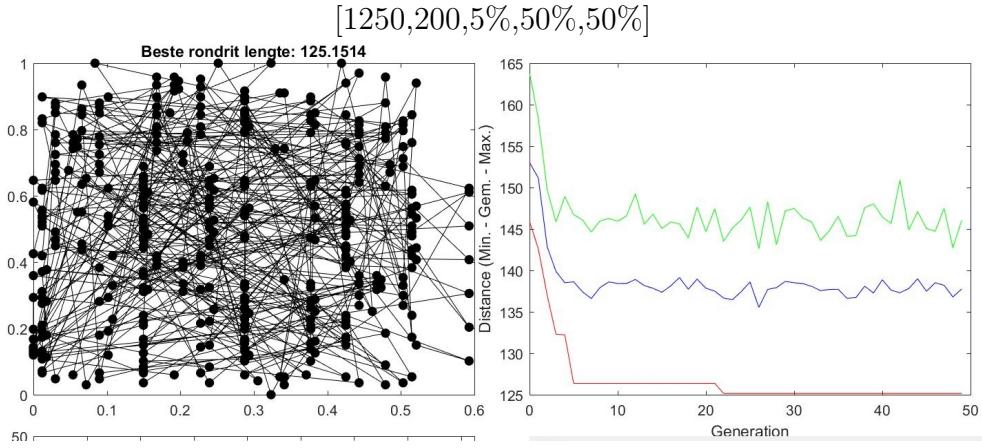
We can observe a slight improvement against the base case, 4 units of distance. Timewise, it is much more demanding that increasing individuals, since the test took 23.73 seconds to end, 3.5 times more than the base case. Any higher number of generations will only hinder the process by making it tediously slow.

As happened in the previous representation tests, the general test for path rep. showed that the value 5% for elitism was without a doubt the best that could be set. So, the last to test is crossover and mutation, and here comes the tricky part for this representation. We found in our general test that no matter what the percentage for mutation or crossover was being tested, the results were almost independant from them. So, given that the base case already tested for a high crossover, and low mutation, the designed tests are, for low crossover and high mutation, and for a 50—50.



Although somewhat better than the base case, the result from one test to the other differ so little that it confirms the result of the general test, stating that, for this representation, given our implementation with our operators, the percentage is not really relevant. But, as it's still better than the base, and the best of the two, the percentages chosen for the combined test are 50%—50%. Obviously, as the number of individuals or generations did not change, the time is almost identical to the base case

For the combined test, in hope to get the best result, this is the test made



Unlike the previous combined test, the big expected improvement is nowhere to be seen. Going from a length of 127 to 120 is just a mere 5.2% of decrease. Furthermore, the combined increase of individuals and generations, given that the former is very high, makes the test to take extremely long compared to the base, going from 6.88 seconds to 141.02 seconds, 20.5 times more. **TODO: Añadir posibles causas de que no mejore, y mencionar cuál puede ser la razón de que no cambie prácticamente nada cuando se modifica mutacion y crossover**

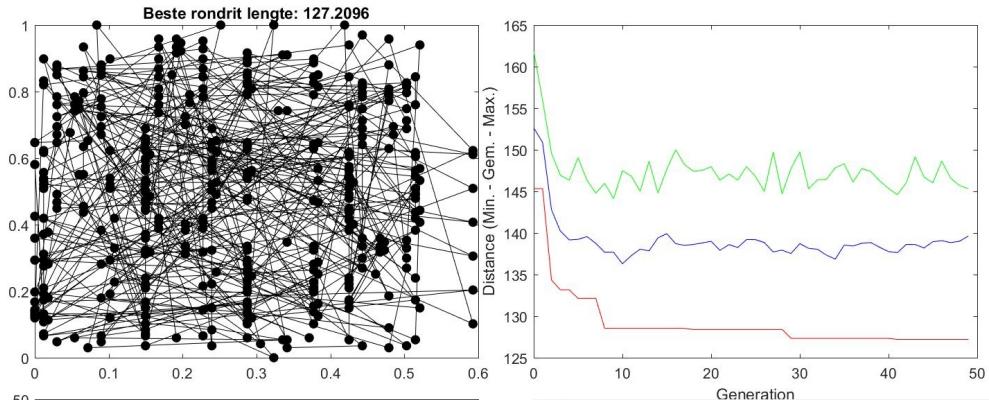
3 Optional part

We opted to implement a new survivor selection strategy. The method selected is round robin tournament, which holds pairwise competitions amongst the whole set of individuals (with our implementation), against q randomly selected, and the selected value for q is 10, as it is typically recommended [1].

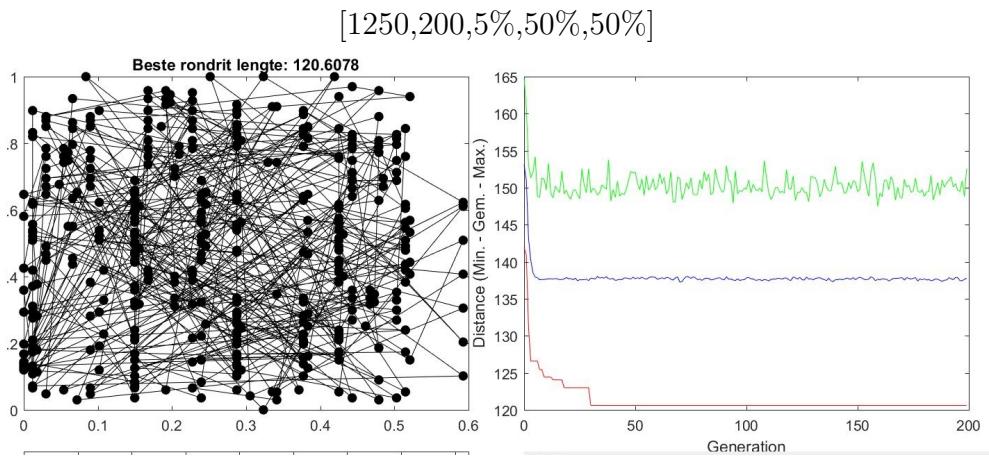
This method consists of holding, in a round robin tournament way, competitions, and keeping the μ **FIX** better individuals. In our implementation, we skipped the round robin tournament, and made a pure

The tests made for this implementation, as it uses the crossover we implemented, we decided to test against the best. As a reminder, here is the result for the base case

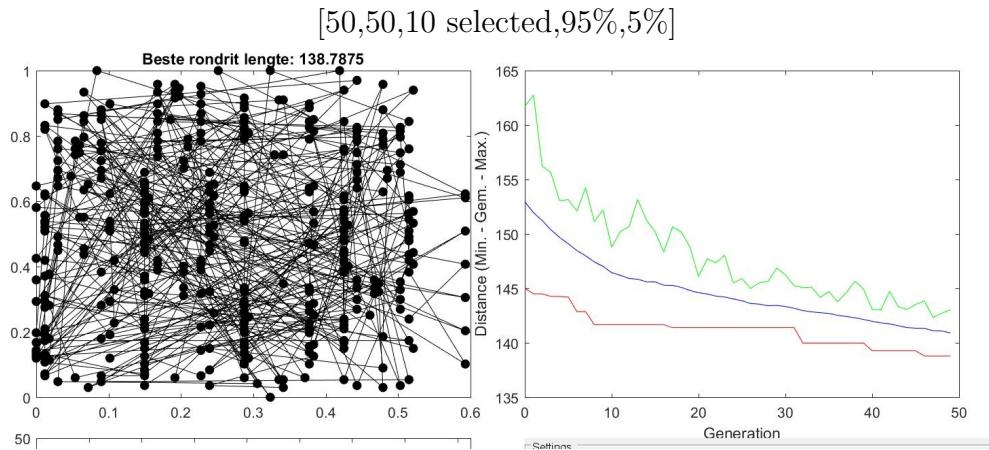
[50,50,5%,95%,5%]



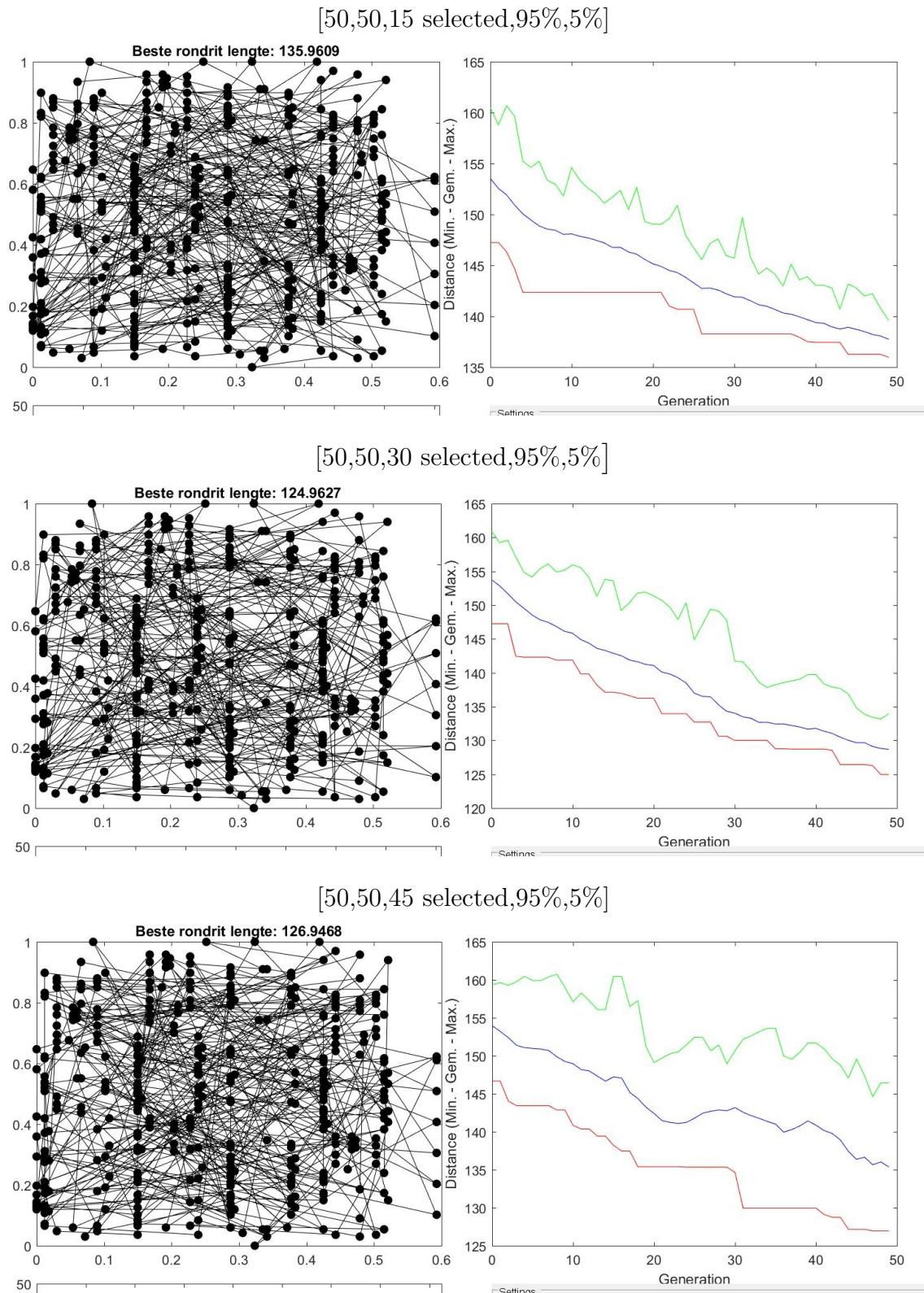
And, the best result obtained after modifying it, with the "best" parameters, according to our general tests

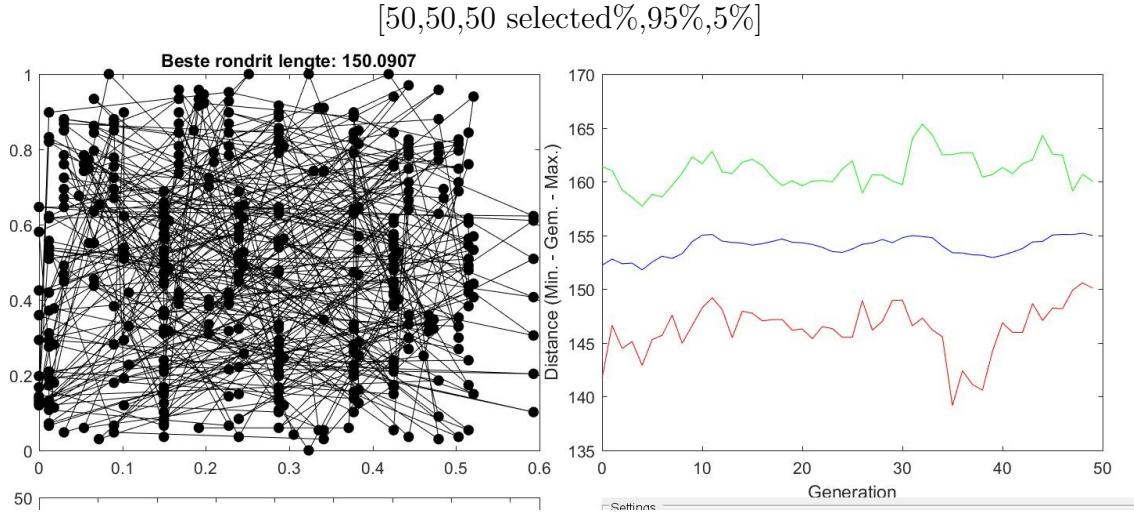


As can be seen, the new selection method started with a worse result, having the same parameters as the base case, only with the % of elitism changed to the number of selected individuals, 10



The only parameter that we can test for is the number of selected individuals from the tournament, and we did so, with 10 (the previous graph), 15,30, 45 and 50





As we can see, with 15 individuals the result is better, but the best outcome comes when the number of individuals is either 30 (the best) or 45 (second best). And, as expected, using all 50 individuals does not only not improve, but actually worsens the result, as it modifies all individuals, no good result is keep from generation to generation. Comparing the time spent on these tests, around 5.5-5.8 each, it is already less time than the average of the base case with elitism, 6.88. Regarding the improvement, when tuning the elitism, the result barely improved at all, but looking at the results with this new selection method, only modifying the number of individuals selected gives us an improvement of 9.96%, which is already more than the "best" solution obtained by tuning all parameters when the selection method was elitism. It could be concluded that this method is better for this problem.

References

- [1] A.E. Eiben, James E Smith. *Introduction to Evolutionary Computing*. Springer
- [2] P. Larrañaga, C.M.H. Kuijpers, R.H. Murga, I. Inza, S. Dizdarevic *Genetic algorithms for the travelling salesman problem: A review of representations and operators*. University of the Basque Country
- [3] Dirk Roose *Genetic algorithms lectures and slides* KU Leuven
- [4] mnemstudio.org *Genetic algorithms mutations* <http://mnemstudio.org/genetic-algorithmsmutation.htm>

4 Appendix

4.1 tsp_ImprovePopulation.m

```
1 % tsp_ImprovePopulation.m
2 % Author: Mike Matton
3 %
4 % This function improves a tsp population by removing local loops
5 % from
6 % each individual.
7 %
8 % Syntax: improvedPopulation = tsp_ImprovePopulation( popsize ,
9 % ncities , pop , improve , dists )
10 %
11 % Input parameters:
12 %   popsize           - The population size
13 %   ncities            - the number of cities
14 %   pop                - the current population (adjacency
15 %                         representation)
16 %   improve             - Improve the population (0 = no improvement
17 %                         , <>0 = improvement)
18 %   dists               - distance matrix with distances between the
19 %                         cities
20 %
21 % Output parameter:
22 %   improvedPopulation - the new population after loop removal (
23 %                         if improve
24 %                                     <> 0 , else the unchanged population).
25 %
26 function newpop = tsp_ImprovePopulation( popsize , ncities , pop ,
27 %                                         improve , dists )
28 %
29 if (improve)
30   for i=1:popsize
31 %
32     result = improve_path( ncities , pop(i,:) , dists );
33 %
34     pop(i,:) = path2adj( result );
35 %
36   end
37 end
38 %
39 newpop = pop;
```

4.2 run_ga.m

```

1 function run_ga(x, y, NIND, MAXGEN, NVAR, SELECTION,
2     STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP, ah1 ,
3     ah2 , ah3)
4 % usage: run_ga(x, y,
5 %                 NIND, MAXGEN, NVAR,
6 %                 SELECTION, STOP_PERCENTAGE,
7 %                 PR_CROSS, PR_MUT, CROSSOVER,
8 %                 ah1 , ah2 , ah3)
9 %
10 %
11 % x, y: coordinates of the cities
12 % NIND: number of individuals
13 % MAXGEN: maximal number of generations
14 %
15 % MODIFIED %%%%%%
16 % SELECTION: if elitism: percentage of elite population , else ,
17 %               it 's mu, quantity of individuals to be selected
18 % MODIFIED %%%%%%
19 %
20 %
21 % STOP_PERCENTAGE: percentage of equal fitness (stop criterium)
22 % PR_CROSS: probability for crossover
23 % PR_MUT: probability for mutation
24 % CROSSOVER: the crossover operator
25 % calculate distance matrix between each pair of cities
26 % ah1 , ah2 , ah3: axes handles to visualise tsp
27 {NIND MAXGEN NVAR SELECTION STOP_PERCENTAGE PR_CROSS PR_MUT
28     CROSSOVER LOCALLOOP};

tic;
if (SELECTION<=1 && SELECTION>=0)
    GGAP = 1 - SELECTION;
end
mean_fits=zeros(1,MAXGEN+1);
worst=zeros(1,MAXGEN+1);
Dist=zeros(NVAR,NVAR);
for i=1:size(x,1)
    for j=1:size(y,1)
        Dist(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
    end
end
% initialize population
Chrom=zeros(NIND,NVAR);

```

```

41 for row=1:NIND
42     %Chrom( row , : )=path2adj( randperm(NVAR) );
43     Chrom( row , : )=randperm(NVAR);
44 end
45 gen=0;
46 % number of individuals of equal fitness needed to stop
47 stopN=ceil(STOP_PERCENTAGE*NIND);
48 % evaluate initial population
49 ObjV = tspfun(Chrom, Dist);
50 best=zeros(1,MAXGEN);
51 % generational loop
52 while gen<MAXGEN
53     sObjV=sort(ObjV);
54     best( gen+1)=min(ObjV);
55     minimum=best( gen+1);
56     mean_fits( gen+1)=mean(ObjV);
57     worst( gen+1)=max(ObjV);
58     for t=1:size(ObjV,1)
59         if (ObjV( t )==minimum)
60             break;
61         end
62     end
63
64 %visualizeTSP (x,y,adj2path(Chrom(t,:)), minimum, ah1,
65 %gen, best, mean_fits, worst, ah2, ObjV, NIND, ah3);
66 visualizeTSP(x,y,Chrom(t,:), minimum, ah1, gen, best,
67 mean_fits, worst, ah2, ObjV, NIND, ah3);
68
69 if (sObjV(stopN)-sObjV(1) <= 1e-15)
70     break;
71 end
72 %assign fitness values to entire population
73 FitnV=ranking(ObjV);
74 %select individuals for breeding
75 if (SELECTION <=1)
76     SelCh=select( 'sus' , Chrom, FitnV, SELECTION);
77 else
78     SelCh=select_rr( 'sus' , Chrom, FitnV, SELECTION);
79 end
80 %recombine individuals (crossover)
81 SelCh = recombin(CROSSOVER, SelCh, PR_CROSS);
82 %SelCh=mutateTSP( 'inversion' , SelCh ,PR_MUT);
83 SelCh=mutateTSP( 'insertion' , SelCh ,PR_MUT); % <-- line
84     changed, now insertion mutation is used
85 %evaluate offspring, call objective function

```

```

83     ObjVSel = tspfun( SelCh , Dist ) ;
84     %reinsert offspring into population
85     [ Chrom , ObjV ]=reins( Chrom , SelCh , 1 , 1 , ObjV , ObjVSel ) ;
86
87     Chrom = tsp_ImprovePopulation( NIND , NVAR , Chrom ,
88                                     LOCALLOOP , Dist ) ;
89     %increment generation counter
90     gen=gen+1;
91
92 end
93 toc ;
94 minimum
end

```

4.3 insertion.m

```

1 % low level function for TSP mutation
2 % Representation is an integer specifying which encoding is used
3 % 1 : adjacency representation
4 % 2 : path representation
5 %
6
7 function NewChrom = insertion( OldChrom )
8
9     NewChrom = OldChrom ;
10    % select two positions in the tour
11    rndi = zeros(1 , 2) ;
12    while rndi(1) == rndi(2)
13        rndi=rand_int(1 , 2 , [1 size(NewChrom , 2)]) ;
14    end
15    rndi = sort(rndi) ;
16
17    % get the value of the first random position
18    temp = NewChrom(rndi(1)) ;
19    % insert this value in the second random position
20    NewChrom = insertAt( NewChrom , temp , rndi(2) ) ;
21    % remove the first random position
22    NewChrom( rndi(1) ) = [] ;
23    % End of function
24
25
26 function arrOut = insertAt( arr , val , index )
27     if index == numel(arr)+1
28         arrOut = [ arr val ] ;

```

```

29     else
30         arrOut = [ arr(1:index-1) val arr(index:end) ];
31     end
32 end

```

4.4 order_crossover.m

```

1 % Syntax: NewChrom = order_crossover(OldChrom, XOVR)
2 %
3 % Input parameters:
4 %   OldChrom - Matrix containing the chromosomes of the old
5 %               population. Each line corresponds to one
6 %               individual
7 %               (in any form, not necessarily real values).
8 %   XOVR      - Probability of recombination occurring between
9 %               pairs
10 %               of individuals.
11 %
12 % Output parameter:
13 %   NewChrom - Matrix containing the chromosomes of the
14 %               population
15 %               after mating, ready to be mutated and/or
16 %               evaluated,
17 %               in the same format as OldChrom.
18 %
19 %
20
21 function NewChrom = order_crossover(OldChrom, XOVR)
22
23 if nargin < 2, XOVR = NaN; end
24 [rows, ~] = size(OldChrom);
25
26 maxrows = rows;
27 if rem(rows, 2) ~= 0
28     maxrows = maxrows - 1;
29 end
30
31 for row = 1:2:maxrows
32
33     % crossover of the two chromosomes
34     % results in 2 offsprings
35     if rand < XOVR      % recombine with a given probability
36         MatrixChrom = order_low_level([OldChrom(row, :); OldChrom(
37             row+1, :)]);
38         NewChrom(row, :) = MatrixChrom(1, :);
39     end
40 end

```

```

33 NewChrom( row+1,:) = MatrixChrom( 2 , : );
34 else
35 NewChrom( row,: ) = OldChrom( row ,: );
36 NewChrom( row+1,: ) = OldChrom( row+1,: );
37 end
38 end
39
40 if rem( rows ,2) ~= 0
41 NewChrom( rows ,:) =OldChrom( rows ,: );
42 end
43
44 % End of function

```

4.5 order_low_level.m

```

1 % low level function for calculating an offspring
2 % given 2 parent in the Parents – agrument
3 % Parents is a matrix with 2 rows , each row
4 % represent the genocode of the parent
5 %
6 % Returns a matrix containing the offspring
7
8
9 function Offspring=order_low_level( Parents )
10
11 cols = size( Parents ,2 );
12
13 Offspring=zeros(2 ,cols );
14
15 start_index = rand_int(1 , 1 , [1 , cols - 1]);
16 end_index = rand_int(1 , 1 , [start_index + 1 , cols ]);
17
18 Offspring (1 , start_index:end_index) = Parents (2 , start_index:
19 end_index);
Offspring (2 , start_index:end_index) = Parents (1 , start_index:
end_index);
20
21
22 for off=1:2
23 Buff = Parents (off ,: );
24 Buff = [ Buff(end_index+1:end) , Buff(1:end_index )];
25
26 members = ismember(Buff , Offspring (off , :));
27 Buff(members == 1) = 0;

```

```

28
29     ii = 1;
30     X = find(Buff);
31     for jj=1:start_index - 1
32         if Buff(X(ii)) ~= 0
33             Offspring(off, jj) = Buff(X(ii));
34             Buff(X(ii)) = 0;
35             ii = mod(ii, cols) + 1;
36         end
37     end
38
39     ii = 1;
40     X = find(Buff);
41     for jj=end_index + 1:cols
42         if Buff(X(ii)) ~= 0
43             Offspring(off, jj) = Buff(X(ii));
44             Buff(X(ii)) = 0;
45             ii = mod(ii, cols) + 1;
46         end
47     end
48     %Offspring(off, end_index+1:end) = Buff(start_index:end);
49     %Offspring(off, 1:start_index - 1) = Buff(1:start_index -
50     1);
51 end
51 % end function

```

4.6 tspgui.m

```

1 Crossover = 'order_crossover';
2 crossover = uicontrol(ph, 'Style', 'popupmenu', 'String', {'',
3     'order_crossover'}, 'Value', 1, 'Position', [10 50 130 20], 'Callback', @crossover_Callback);

```

4.7 tspfun.m

```

1 %
2 % ObjVal = tspfun(Phen, Dist)
3 % Implementation of the TSP fitness function
4 % Phen contains the phenocode of the matrix coded in path
5 % representation
6 % Dist is the matrix with precalculated distances between each
    pair of cities

```

```

7 % ObjVal is a vector with the fitness values for each candidate
8 % tour
9 %
10
11 function ObjVal = tspfun(Phen, Dist)
12 % the objective function works with adjacency representation.
13 % In this
14 % version , path representation is used , so the fitness
15 % function should
16 % be adapted. Now, the phenotype is converted to adjacency
17 % representation first , and then , the Objective Value is
18 % computed as it
19 % was computed in the original version .
20 adj = zeros(size(Phen));
21 for row=1:size(Phen)
22     adj(row,:) = path2adj(Phen(row,:));
23 end
24
25
26
27 % End of function

```

4.8 mutateTSP.m

```

1 % MUTATETSP.M          (MUTATION for TSP high-level function)
2 %
3 % This function takes a matrix OldChrom containing the
4 % representation of the individuals in the current population ,
5 % mutates the individuals and returns the resulting population .
6 %
7 % Syntax: NewChrom = mutate(MUTF, OldChrom, MutOpt)
8 %
9 % Input parameter:
10 %    MUTF      – String containing the name of the mutation
11 %              function
12 %    OldChrom – Matrix containing the chromosomes of the old
13 %              population . Each line corresponds to one
14 %              individual .
15 %    MutOpt   – mutation rate
16 %
17 % Output parameter:

```

```

15 % NewChrom - Matrix containing the chromosomes of the
16 % population
17 % after mutation in the same format as OldChrom.
18
19 function NewChrom = mutateTSP(MUT_F, OldChrom, MutOpt)
20
21 % Check parameter consistency
22 if nargin < 2, error('Not_enough_input_parameters'); end
23
24 [rows, ~] = size(OldChrom);
25 NewChrom=OldChrom;
26
27 for r=1:rows
28   if rand<MutOpt
29     NewChrom(r,:)= feval(MUT_F, OldChrom(r,:));
30   end
31 end
32
33 % End of function

```

4.9 run_ga_test.m

```

1 function minimum=run_ga_test(modality, x, y, NIND, MAXGEN, NVAR,
2   SELECTION, ...
3   STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP)
4 % usage: run_ga(x, y,
5 %                 NIND, MAXGEN, NVAR,
6 %                 ELITIST, STOP_PERCENTAGE,
7 %                 PR_CROSS, PR_MUT, CROSSOVER,
8 %                 ah1, ah2, ah3)
9 %
10 % x, y: coordinates of the cities
11 % NIND: number of individuals
12 % MAXGEN: maximal number of generations
13 % ELITIST: percentage of elite population
14 % STOP_PERCENTAGE: percentage of equal fitness (stop criterium)
15 % PR_CROSS: probability for crossover
16 % PR_MUT: probability for mutation
17 % CROSSOVER: the crossover operator
18 % calculate distance matrix between each pair of cities
19 % ah1, ah2, ah3: axes handles to visualise tsp

```

```

20 { modality NIND MAXGEN NVAR SELECTION STOP_PERCENTAGE PR_CROSS
21 PR_MUT CROSSOVER LOCALLOOP};

22
23 if (SELECTION<=1 && SELECTION>=0)
24     GGAP = 1 - SELECTION;
25 end

26
27 mean_fits=zeros(1,MAXGEN+1);
28 worst=zeros(1,MAXGEN+1);
29 Dist=zeros(NVAR,NVAR);
30 for i=1:size(x,1)
31     for j=1:size(y,1)
32         Dist(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
33     end
34 end
35 % initialize population
36 Chrom=zeros(NIND,NVAR);
37 for row=1:NIND
38     if (modality==1)
39         Chrom(row,:)=randperm(NVAR);
40     else
41         Chrom(row,:)=path2adj(randperm(NVAR));
42     end
43 end
44 gen=0;
45 % number of individuals of equal fitness needed to stop
46 stopN=ceil(STOP_PERCENTAGE*NIND);
47 % evaluate initial population
48 ObjV = tspfun(Chrom, Dist);
49 best=zeros(1,MAXGEN);
50 % generational loop
51 while gen<MAXGEN
52     sObjV=sort(ObjV);
53     best(gen+1)=min(ObjV);
54     minimum=best(gen+1);
55     mean_fits(gen+1)=mean(ObjV);
56     worst(gen+1)=max(ObjV);
57     for t=1:size(ObjV,1)
58         if (ObjV(t)==minimum)
59             break;
60         end
61     end
62 end
63

```

```

64 %visualizeTSP (x ,y ,adj2path (Chrom(t ,:)) , minimum , ah1 ,
65   gen , best , mean_fits , worst , ah2 , ObjV , NIND , ah3 );
66
67 if (sObjV(stopN)-sObjV(1) <= 1e-15)
68   break;
69 end
70 %assign fitness values to entire population
71 FitnV=ranking(ObjV);
72 %select individuals for breeding
73 if (SELECTION <=1)
74   SelCh=select( 'sus' , Chrom , FitnV , SELECTION);
75 else
76   SelCh=select_rr( 'sus' , Chrom , FitnV , SELECTION);
77 end
78
79 %recombine individuals (crossover)
80 SelCh = recombin(CROSSOVER,SelCh,PR_CROSS);
81
82 if (modality==1)
83   SelCh=mutateTSP( 'insertion' ,SelCh,PR_MUT);
84 else
85   SelCh=mutateTSP( 'inversion' ,SelCh,PR_MUT);
86 end
87
88 %evaluate offspring , call objective function
89 ObjVSel = tspfun(SelCh,Dist);
90 %reinsert offspring into population
91 [Chrom , ObjV]=reins(Chrom ,SelCh ,1 ,1 ,ObjV ,ObjVSel);
92
93 Chrom = tsp_ImprovePopulation(NIND , NVAR , Chrom ,
94   LOCALLOOP ,Dist);
95 %increment generation counter
96 gen=gen+1;
end
end

```

4.10 tspgui_test.m

```

1 %Change in lines 49-75, and 106
2
3 function tspgui_test(CROSSOVER, NIND, MAXGEN, ELITIST ,PR_CROSS,
4   PR_MUT,LOCALLOOP, testName)
5

```

```

6 %
7 %NIND=50;    % Number of individuals
8 %MAXGEN=100;   % Maximum no. of generations
9 NVAR=26;      % No. of variables
10 PRECI=1;     % Precision of variables
11 %ELITIST=0.05;    % percentage of the elite population
12 GGAP=1-ELITIST;   % Generation gap
13 STOPPERCENTAGE=.95;    % percentage of equal fitness individuals
14 %           for stopping
15 %PR_CROSS=.95;    % probability of crossover
16 %PR_MUT=.05;      % probability of mutation
17 %LOCALLOOP=1;     % local loop removal
18 %Crossover = 'xalt_edges'; % default crossover operator
19 %
20 %
21 %
22 %
23 %
24 %
25 %
26 %
27 %
28 %
29 %
30 %
31 %
32 %
33 %
34 %
35 %
36 %
37 %
38 %
39 %
40 %
41 %
42 %
43 %
44 %

```

```

45 %      x=XY(:,1);
46 %      y=XY(:,2);
47 %end
48
49 % load the data sets from the benchmark list
50 benchmarks = dir('TSPBenchmark/');
51 datasets=cell( size(benchmarks,1)-2,1);
52 for i=1:size(datasets,1)
53     datasets{i} = benchmarks(i+2).name;
54 end
55
56 % start with first dataset
57 data = load(['TSPBenchmark/bcl380.tsp']);
58 x=data(:,1)/max([data(:,1);data(:,2)]);
59 y=data(:,2)/max([data(:,1);data(:,2)]);
60 NVAR=size(data,1);
61
62 %datasets
63
64 % initialise the user interface
65 fh = figure('Visible','off','Name','TSP-Tool','Position',
66             [0,0,1024,768]);
66 ah1 = axes('Parent',fh,'Position',[.1 .55 .4 .4]);
67 plot(x,y,'ko')
68 ah2 = axes('Parent',fh,'Position',[.55 .55 .4 .4]);
69 axes(ah2);
70 xlabel('Generation');
71 ylabel('Distance_(Min. -- Gem. -- Max.)');
72 ah3 = axes('Parent',fh,'Position',[.1 .1 .4 .4]);
73 axes(ah3);
74 title('Histogram');
75 xlabel('Distance');
76 ylabel('Number');
77
78 ph = uipanel('Parent',fh,'Title','Settings','Position',[.55 .05
79               .45 .45]);
80 datasetpopuptxt = uicontrol(ph,'Style','text','String','Dataset',
81                             'Position',[0 260 130 20]);
80 datasetpopup = uicontrol(ph,'Style','popupmenu','String',datasets,
81                          'Value',1,'Position',[130 260 130 20],'Callback',
82                          @datasetpopup_Callback);
81 llooppopuptxt = uicontrol(ph,'Style','text','String','Loop-
82 Detection','Position',[260 260 130 20]);
82 llooppopup = uicontrol(ph,'Style','popupmenu','String',{ 'off ','on '},
83                         'Value',1,'Position',[390 260 50 20],'Callback',

```

```

    @llooppopup_Callback);
83 ncitiesslidertxt = uicontrol(ph, 'Style', 'text', 'String', '#_Cities',
     , 'Position', [0 230 130 20]);
84 %ncitiesslider = uicontrol(ph, 'Style', 'slider', 'Max', 128, 'Min', 4,
     Value', NVAR, 'Sliderstep', [0.012 0.05], 'Position', [130 230 150
     20], 'Callback', @ncitiesslider_Callback);
85 ncitiessliderv = uicontrol(ph, 'Style', 'text', 'String', NVAR,
     Position', [280 230 50 20]);
86 nindslidertxt = uicontrol(ph, 'Style', 'text', 'String', '#_
     Individuals', 'Position', [0 200 130 20]);
87 nindslider = uicontrol(ph, 'Style', 'slider', 'Max', 1000, 'Min', 10,
     Value', NIND, 'Sliderstep', [0.001 0.05], 'Position', [130 200 150
     20], 'Callback', @nindslider_Callback);
88 nindsliderv = uicontrol(ph, 'Style', 'text', 'String', NIND, 'Position'
     ,[280 200 50 20]);
89 genslidertxt = uicontrol(ph, 'Style', 'text', 'String', '#_Generations
     , 'Position', [0 170 130 20]);
90 genslider = uicontrol(ph, 'Style', 'slider', 'Max', 1000, 'Min', 10,
     Value', MAXGEN, 'Sliderstep', [0.001 0.05], 'Position', [130 170 150
     20], 'Callback', @genslider_Callback);
91 gensliderv = uicontrol(ph, 'Style', 'text', 'String', MAXGEN, 'Position
     ,[280 170 50 20]);
92 mutslidertxt = uicontrol(ph, 'Style', 'text', 'String', 'Pr._Mutation'
     , 'Position', [0 140 130 20]);
93 mutslider = uicontrol(ph, 'Style', 'slider', 'Max', 100, 'Min', 0, 'Value
     , round(PR MUT*100), 'Sliderstep', [0.01 0.05], 'Position', [130
     140 150 20], 'Callback', @mutslider_Callback);
94 mutsliderv = uicontrol(ph, 'Style', 'text', 'String', round(PR MUT
     *100), 'Position', [280 140 50 20]);
95 crossslidertxt = uicontrol(ph, 'Style', 'text', 'String', 'Pr._
     Crossover', 'Position', [0 110 130 20]);
96 crossslider = uicontrol(ph, 'Style', 'slider', 'Max', 100, 'Min', 0,
     Value', round(PR CROSS*100), 'Sliderstep', [0.01 0.05], 'Position'
     ,[130 110 150 20], 'Callback', @crossslider_Callback);
97 crosssliderv = uicontrol(ph, 'Style', 'text', 'String', round(PR CROSS
     *100), 'Position', [280 110 50 20]);
98 elitslidertxt = uicontrol(ph, 'Style', 'text', 'String', '%elite', ,
     Position', [0 80 130 20]);
99 elitslider = uicontrol(ph, 'Style', 'slider', 'Max', 100, 'Min', 0,
     Value', round(ELITIST*100), 'Sliderstep', [0.01 0.05], 'Position'
     ,[130 80 150 20], 'Callback', @elitslider_Callback);
100 elitsliderv = uicontrol(ph, 'Style', 'text', 'String', round(ELITIST
     *100), 'Position', [280 80 50 20]);
101 crossover = uicontrol(ph, 'Style', 'popupmenu', 'String', {
     xalt_edges'}, 'Value', 1, 'Position', [10 50 130 20], 'Callback',
```

```

    @crossover_Callback);
102 %inputbutton = uicontrol(ph,'Style','pushbutton','String','Input
      ','Position',[55 10 70 30],'Callback',@inputbutton_Callback);
103 runbutton = uicontrol(ph,'Style','pushbutton','String','START',
      'Position',[0 10 50 30],'Callback',@runbutton_Callback);

104 set(fh,'Visible','on');

105 %%%
106 %%Added for the tests
107 runbutton_Callback;
108 saveas(fh,testName,'jpg');
109 close(fh);
110 %%%
111 %%%
112 %%%
113
114
115 function datasetpopup_Callback(hObject, eventdata)
116     dataset_value = get(hObject,'Value');
117     dataset = datasets{dataset_value};
118     % load the dataset
119     data = load(['datasets/' dataset]);
120     x=data(:,1)/max([data(:,1);data(:,2)]);y=data(:,2)/max([
121         data(:,1);data(:,2)]);
122     %x=data(:,1);y=data(:,2);
123     NVAR=size(data,1);
124     set(ncitiessliderv,'String',size(data,1));
125     axes(ah1);
126     plot(x,y,'ko')
127 end
128 function llooppopup_Callback(hObject, eventdata)
129     lloop_value = get(hObject,'Value');
130     if lloop_value==1
131         LOCALLOOP = 0;
132     else
133         LOCALLOOP = 1;
134     end
135 function ncitiesslider_Callback(hObject, eventdata)
136     fslider_value = get(hObject,'Value');
137     slider_value = round(fslider_value);
138     set(hObject,'Value',slider_value);
139     set(ncitiessliderv,'String',slider_value);
140     NVAR = round(slider_value);
141 end
142 function nindslider_Callback(hObject, eventdata)

```

```

143 fslider_value = get(hObject, 'Value');
144 slider_value = round(fslider_value);
145 set(hObject, 'Value', slider_value);
146 set(nindsliderv, 'String', slider_value);
147 NIND = round(slider_value);
148 end
149 function genslider_Callback(hObject, eventdata)
150     fslider_value = get(hObject, 'Value');
151     slider_value = round(fslider_value);
152     set(hObject, 'Value', slider_value);
153     set(gensliderv, 'String', slider_value);
154     MAXGEN = round(slider_value);
155 end
156 function mutslider_Callback(hObject, eventdata)
157     fslider_value = get(hObject, 'Value');
158     slider_value = round(fslider_value);
159     set(hObject, 'Value', slider_value);
160     set(mutsliderv, 'String', slider_value);
161     PRMUT = round(slider_value)/100;
162 end
163 function crossslider_Callback(hObject, eventdata)
164     fslider_value = get(hObject, 'Value');
165     slider_value = round(fslider_value);
166     set(hObject, 'Value', slider_value);
167     set(crosssliderv, 'String', slider_value);
168     PR_CROSS = round(slider_value)/100;
169 end
170 function elitslider_Callback(hObject, eventdata)
171     fslider_value = get(hObject, 'Value');
172     slider_value = round(fslider_value);
173     set(hObject, 'Value', slider_value);
174     set(elitsliderv, 'String', slider_value);
175     ELITIST = round(slider_value)/100;
176     GGAP = 1-ELITIST;
177 end
178 function crossover_Callback(hObject, eventdata)
179     crossover_value = get(hObject, 'Value');
180     crossovers = get(hObject, 'String');
181     CROSSOVER = crossovers(crossover_value);
182     CROSSOVER = CROSSOVER{1};
183 end
184 function runbutton_Callback(hObject, eventdata)
185 %set(ncitiesslider, 'Visible', 'off');
186 set(nindslider, 'Visible', 'off');
187 set(genslider, 'Visible', 'off');

```

```

188 set( mutslider , 'Visible' , 'off' );
189 set( crossslider , 'Visible' , 'off' );
190 set( elitslider , 'Visible' , 'off' );
191 run_ga(x, y, NIND, MAXGEN, NVAR, ELITIST, STOP PERCENTAGE,
        PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP, ah1, ah2, ah3)
        ;
192 end_run();
193 end
194 function inputbutton_Callback(hObject, eventdata)
195 [x y] = input_cities(NVAR);
196 axes(ah1);
197 plot(x,y,'ko')
198 end
199 function end_run()
200 %set( ncitiesslider , 'Visible' , 'on' );
201 set( nindslder , 'Visible' , 'on' );
202 set( genslider , 'Visible' , 'on' );
203 set( mutslider , 'Visible' , 'on' );
204 set( crossslider , 'Visible' , 'on' );
205 set( elitslider , 'Visible' , 'on' );
206 end
207 end

```

4.11 test.m

```

1 %%%
2 %{
3 Modifications:
4
5 run_ga_test --> Now returns a value, minimum, does not use
6 the function visualizeTSP (line ~56), and takes into account which
7 crossover operator is used, and whether the selection for the new
     offspring
8 is elitism or roundrobin tournament
9
10 tspgui_test --> Now runs automatically the test with the benchmark
     bcl380
11 configuration, saves and closes the figure
12
13 test --> There are 2 sets of tests. First, specific tests, limited
     in
14 number, made with a purpose. Those tests are run with tspgui-test,
     using
15 the dataset from the benchmark bcl380, and saving the figure.

```

```

16 The second set is the generalistic tests , which check the 4
17 different
18 components to be changed (aka: number of individuals , number of
19 generations , elitism , and percentage of crossover and mutation) ,
20 with all
21 the datasets. Each test is made with a certain number of values ,
22 repeated *reps* (variable in the code , currently set to 5) times ,
23 and the
24 mean obtained by that is the value we use for the plots .
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
%
```

%} %} %}

%Clear previous existing variables

clear

%Check if folder tests exists , if not , create it

if ~**exist**('tests' , 'dir')

mkdir('tests');

end

%Options for Crossover (depending on the crossover , a

 representation will

%be chosen

%CROSSOVER = 'order_crossover';

%CROSSOVER = 'xalt_edges';

% 1 for specific tests (limited tests , with graph saving)

% 2 for benchmark (obtain just the distance)

% 3 for both

perform_tests(3 , CROSSOVER);

perform_optest();

function **perform_tests(n , CROSSOVER)**

if ~**exist**(strcat('tests/' ,CROSSOVER) , 'dir')

mkdir(strcat('tests/' ,CROSSOVER));

end

%Specific test

if (n==1 || n==3)

```

57
58 %Depending on the representation (crossover choosen), the
59 % data
60 % for the specific test is choosen
61 if (strcmp(CROSSOVER, 'order_crossover'))%
62 %
63 NIND= [50,200,1000,1250,50,50,50,1250];
64 MAXGEN= [50,50,50,50,200,50,50,200];
65 ELITIST=0.05;
66 PR_CROSS=[.95,.95,.95,.95,.95,.1,.5,.5,.5];
67 PR_MUT=[.05,.05,.05,.05,.05,.9,.5,.5,.5];
68 LOCALLOOP=1;
69 %
70
71 else
72 %
73 %
74 NIND= [50,200,50,50,50,50,50,50,200,200];
75 MAXGEN= [50,50,200,750,1000,50,50,50,200,200];
76 ELITIST=0.05;
77 PR_CROSS=[.95,.95,.95,.95,.95,.5,.20,.80,.95,.5];
78 PR_MUT=[.05,.05,.05,.05,.05,0.5,.8,.2,.05,.5];
79 LOCALLOOP=1;
80 %
81
82 end
83 %
84 %
85 upbound = size(NIND);
86 for i=1:upbound(2)
87     tspgui_test(CROSSOVER, NIND(i), MAXGEN(i), ELITIST(i),
88     ...
89     PR_CROSS(i),PR_MUT(i),LOCALLOOP, ...

```

```

90      strcat( 'tests/' , Crossover, '/ ', 'general_ ',num2str
91          ( i ) );
92
93 end %end if (n==1 || n==3)
94
95 %General tests
96 if (n==2 || n==3)
97
98     reps = 5;
99
100    %
101    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
102
103    NIND=[50,100,150,200,500,750,1000];
104    MAXGEN=[50,100,150,200,500,750,1000];
105    ELITIST=[0,0.05, 0.1, 0.25, 0.5, 0.75, 1] ;
106    STOP_PERCENTAGE=.95;
107    PR_CROSS=[0, 0.05, 0.1, 0.25, 0.5, 0.75, 1];
108    PR_MUT=[1,0.95, 0.9, 0.75, 0.5, 0.25, 0];
109    LOCALLOOP=1;
110    %
111    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
112
113    datasetslist = dir( 'datasets/' );
114    datasets=cell( size( datasetslist ,1)-2 ,1);
115
116    upbound = size(NIND);
117    modality = 0;
118
119    if (strcmp(Crossover, 'order_crossover'))
120        modality=1;
121    end
122
123    %possible values for each test, city configurations,# of
124    %different tests
125    values = zeros(upbound(2),size( datasets ,1),4);
126
127    for j=1:upbound(2)
128        for i=1:size( datasets ,1)
129
130            datasets{i} = datasetslist(i+2).name;
131            data = load([ 'datasets/' datasets{i}]);
132            x=data(:,1)/max([ data(:,1); data(:,2) ]);

```

```

129 y=data (: , 2) /max([ data (: , 1) ; data (: , 2) ]) ;
130 NVAR=size(data , 1) ;
131
132
133 %Tests for number of indv
134 temp = zeros(reps , 1) ;
135 for n=1:reps
136     temp(n)=run_ga_test(modality , x,y,NIND(j) ,
137                         MAXGEN(1) , ...
138                         NVAR, ELITIST(1) , ...
139                         STOP_PERCENTAGE, PR_CROSS(1) ,
140                         PR_MUT(1) , ...
141                         CROSSOVER,LOCALLOOP ) ;
142
143
144 %Tests for number of generations
145 temp = zeros(reps , 1) ;
146 for n=1:reps
147     temp(n)=run_ga_test(modality ,x,y,NIND(1) , ...
148                         MAXGEN(j) ,NVAR, ELITIST(1) , ...
149                         STOP_PERCENTAGE, PR_CROSS(1) ,
150                         PR_MUT(1) , ...
151                         CROSSOVER,LOCALLOOP ) ;
152
153
154 %Tests for elitism
155 temp = zeros(reps , 1) ;
156 for n=1:reps
157     temp(n)=run_ga_test(modality ,x,y,NIND(1) ,
158                         MAXGEN(1) , ...
159                         NVAR, ELITIST(j) , ...
160                         STOP_PERCENTAGE, PR_CROSS(1) ,
161                         PR_MUT(1) , ...
162                         CROSSOVER,LOCALLOOP ) ;
163
164
165 %Tests for % of crossover and mutation
166 temp = zeros(reps , 1) ;
167 for n=1:reps
168     temp(n)=run_ga_test(modality ,x,y,NIND(1) ,
169                         MAXGEN(1) , ...

```

```

168 NVAR, ELITIST(1) , ...
169 STOP_PERCENTAGE, PR_CROSS(j) ,
170 PR_MUT(j) ,...
171 CROSSOVER, LOCALLOOP );
172
173
174
175 end %for i=1:size(datasets,1);
176 end %for n=1,size(NIND)
177
178
179 %Plots
180 %Number of indv plot
181 nindF = figure;
182 p = plot(NIND, values(:,:,1));
183 xlabel('#of Individuals');
184 title('Increase_of_number_of_individuals_(all_datasets)');
185 ylabel('TSP_distance');
186 set(gca, 'XTick', NIND);
187 saveas(nindF, strcat('tests/','Crossover', '/numberIndiv'),
188 'jpg');
189 close(nindF);
190
191 %Number of gen plot
192 genF = figure;
193 p = plot(NIND, values(:,:,2));
194 xlabel('#of_Generations');
195 title('Increase_of_generations_(all_datasets)');
196 ylabel('TSP_distance');
197 set(gca, 'XTick', MAXGEN);
198 saveas(genF, strcat('tests/','Crossover', '/numberGens'),
199 'jpg');
200 close(genF);
201
202 %Elitism plot
203 elitF = figure;
204 p = plot(ELITIST, values(:,:,3));
205 xlabel('%of_elitism');
206 title('Increase_of_elitism_(all_datasets)');
207 ylabel('TSP_distance');
208 set(gca, 'XTick', ELITIST);
209 saveas(elitF, strcat('tests/','Crossover', '/elitism'),
210 'jpg');
211 close(elitF);

```

```

209
210 %Crossover/mutation plot
211 porcF = figure;
212 p = plot([1:size(PR_CROSS,2)], values(:,:,4));
213 xlabel('%of_Crossover|Mutation');
214 title('Percentage_of_Crossover_and_mutation');
215 ylabel('TSP_distance');
216 xlab = {'0|1', '0.05|0.95', '0.1|0.9', '0.25|0.75', ...
217 '0.5|0.5',...
218 '0.75|0.25', '1|0'};
219 set(gca, 'XLim',[1 size(PR_CROSS,2)], 'XTick',1:size(
220 PR_CROSS,2) ...
221 , 'XTickLabel',xlab)
222 saveas(porcF, strcat('tests/',CROSSOVER, '/crossMut'), ...
223 jpg);
224 close(porcF);

225
226 end %if (n==2 || n==3)
227 end %end of function

228
229 function perform_optest()
230
231 %Testing optional
232 datasetslist = dir('datasets/');
233 datasets=cell(size(datasetslist,1)-2,1);
234 for i=1:size(datasets,1)
235 datasets{i} = datasetslist(i+2).name;
236 end
237 data = load(['datasets/' datasets{1}]);
238 x=data(:,1)/max([data(:,1);data(:,2)]);
239 y=data(:,2)/max([data(:,1);data(:,2)]);
240
241 NVAR=size(data,1);
242 NIND=50; % Number of individuals
243 MAXGEN=100; % Maximum no. of generations
244 NVAR=26; % No. of variables
245 SELECTION=15; % Number of indv to be selected after
246 tournament
247 STOP_PERCENTAGE=.95; % percentage of equal fitness
248 individuals for stopping
249 PR_CROSS=.95; % probability of crossover
250 PR_MUT=.05; % probability of mutation
251 LOCALLOOP=0; % local loop removal
252 CROSSOVER = 'order_crossover'; % default crossover operator

```

```

249 val = run_ga_test(1,x,y,NIND, MAXGEN, NVAR, SELECTION,
250 STOP_PERCENTAGE, ...
251 PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP);
252 display(val);
253 end

```

4.12 select_rr.m

```

1 % Modified SELECT.M, now uses round robin tournament instead of
2 % elitism
3 %
4 % Syntax: SelCh = select_rr(SEL_F, Chrom, FitnV, GGAP)
5 %
6 % Input parameters:
7 % SEL_F      - Name of the selection function
8 % Chrom       - Matrix containing the individuals (parents) of
9 %                 the current
10 %                  population. Each row corresponds to one
11 % individual.
12 % FitnV      - Column vector containing the fitness values of
13 %                 the
14 %                  individuals in the population.
15 % GGAP        - (optional) Rate of individuals to be selected
16 %                 if omitted 1.0 is assumed
17 %
18 % Output parameters:
19 % SelCh       - Matrix containing the selected individuals.
20 %
21 function SelCh = select_rr(SEL_F, Chrom, FitnV, mu)
22 %
23 % Check parameter consistency
24 if nargin < 3, error('Not_enough_input_parameter'); end
25 %
26 % Identify the population size (Nind)
27 [NindCh, ~] = size(Chrom);
28 [NindF, VarF] = size(FitnV);
29 if NindCh ~= NindF, error('Chrom_and_FitnV_disagree'); end
30 if VarF ~= 1, error('FitnV_must_be_a_column_vector'); end
31 %
32 Nind = NindCh;
33 %
34 if nargin < 4, mu = 10; end
35 %
36 if nargin > 3

```

```

33 if isempty(mu) , mu = 10;
34 elseif isnan(mu) , mu = 10;
35 elseif length(mu) ~= 1, error( 'Mu must be a scalar' );
36 elseif (mu < 9), error( 'Mu must be a scalar bigger
37 than 9' );
38 end
39
40 %Select q random individuals
41 q = zeros(10,1);
42 temp = zeros(10,1);
43 for i=1:10
44     ind = randi(size(Chrom,1)) ;
45
46     %Avoid repetition
47     while(any(ind==temp))
48         ind=randi(size(Chrom,1)) ;
49     end
50
51     temp(i) = ind;
52     q(i) = FitnV(ind);
53 end
54
55
56 %Tournament variable, only index and amount of wins is stored
57 tournament = zeros(size(Chrom,1),2);
58 %Tournament-> every indv compites against q rivals
59 for i=1:size(Chrom,1)
60     tournament(i,1) = i;
61     for j=1:size(q,1)
62         if(FitnV(i) > q(j))
63             tournament(i,2) = tournament(i,2) + 1;
64         end
65     end
66 end
67
68 %Sort tournament by amount of wins
69 sortedTournament = sortrows(tournament,2);
70
71 %Indexes of the mu better individuals
72 indexes = sortedTournament(size(
73     sortedTournament,1)-mu:size(sortedTournament,1),1);
74
75
76 SelCh = [];

```

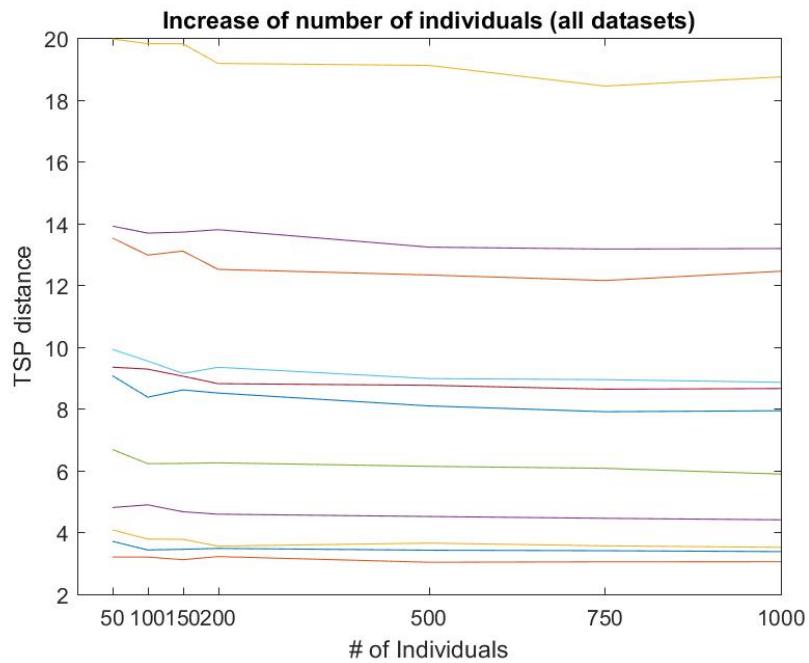
```

77 FitnVSub = FitnV(indexes);
78 ChrIx=feval(SEL_F, FitnVSub, mu);
79 SelCh=[SelCh; Chrom(ChrIx,:)];
80
81 % End of function

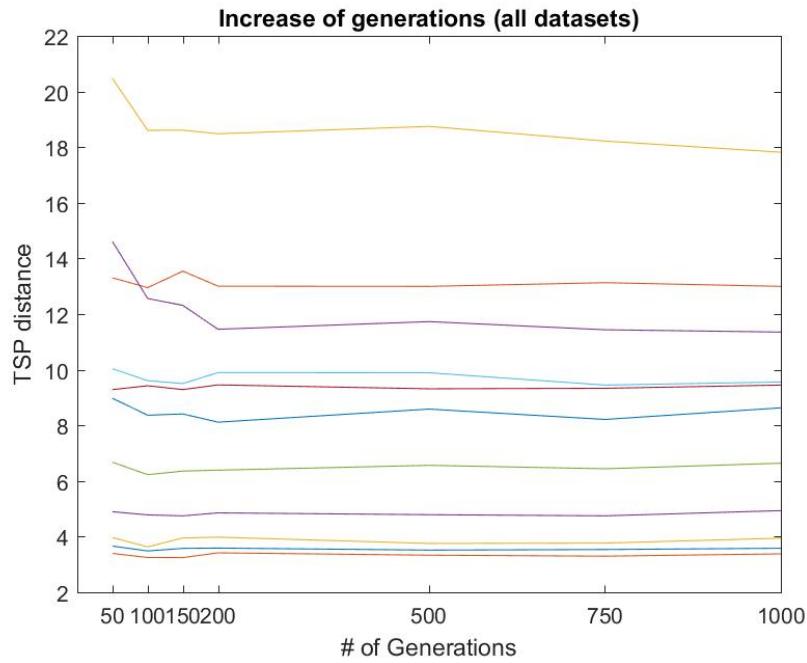
```

4.13 Graph

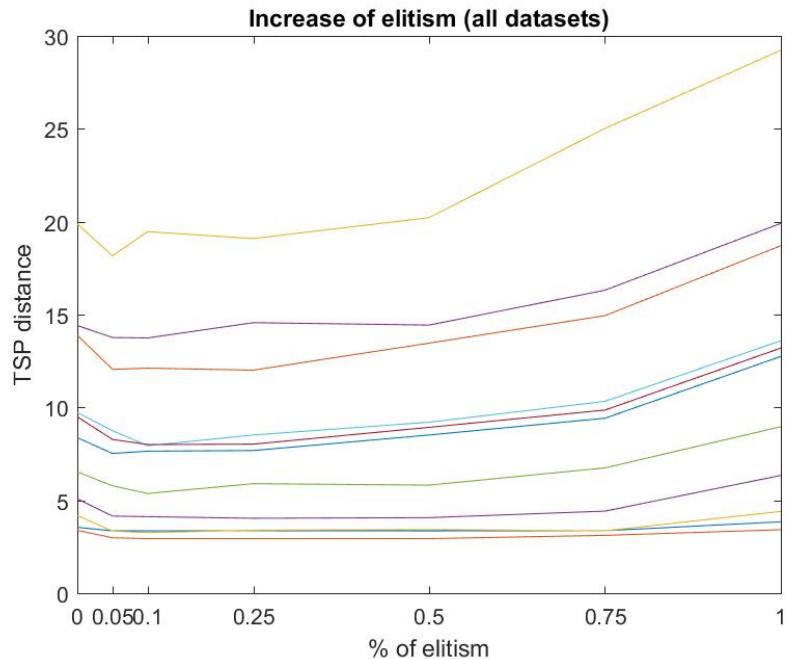
4.13.1 Original-General-Individuals



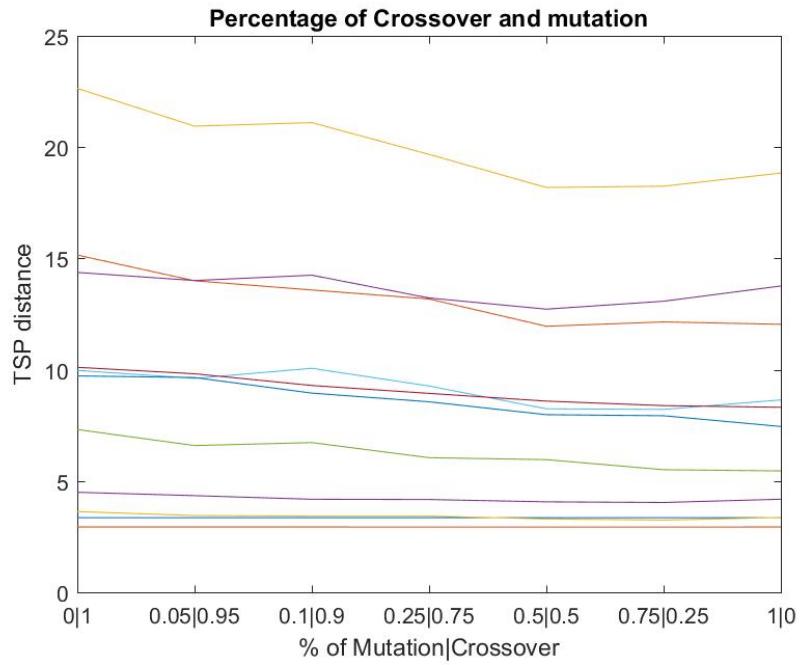
4.13.2 Original-General-Generations



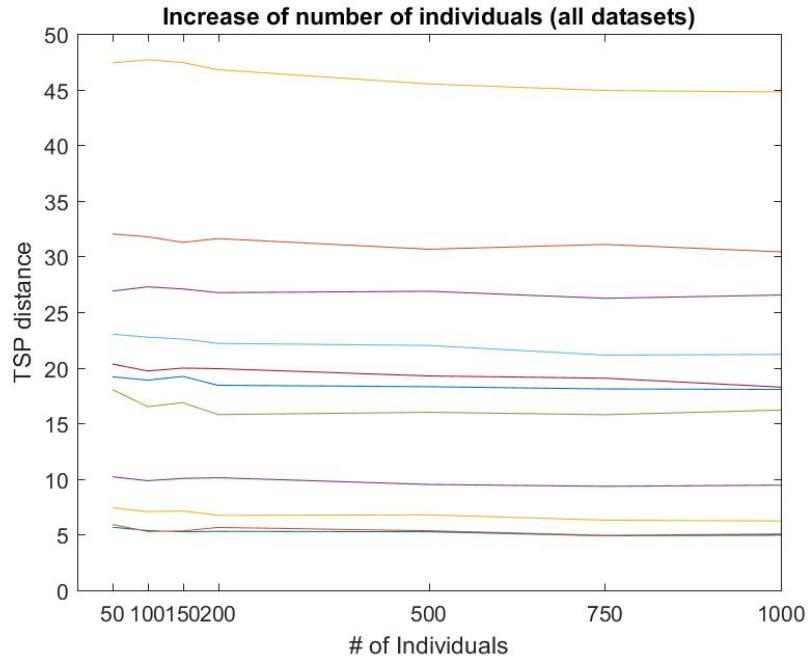
4.13.3 Original-General-Elitism



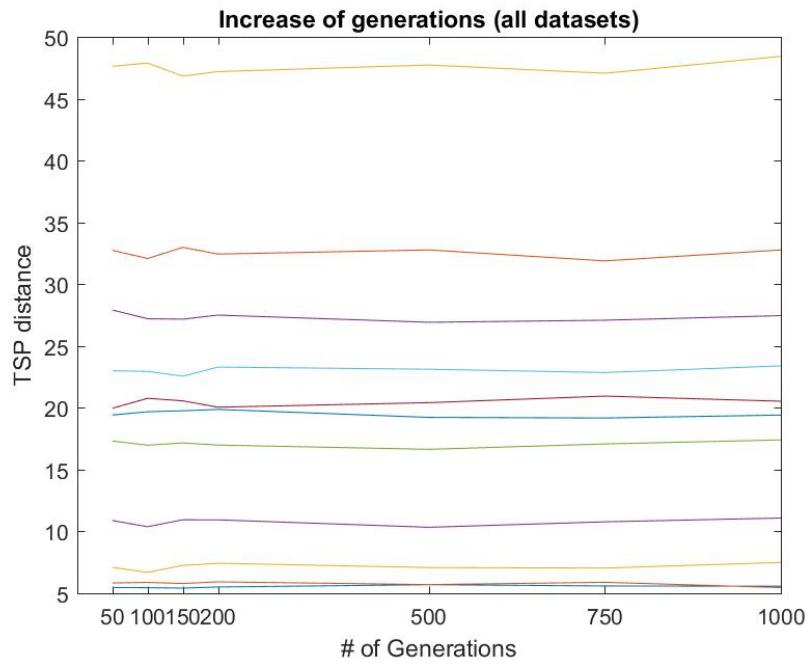
4.13.4 Original-General-Crossover and mutation



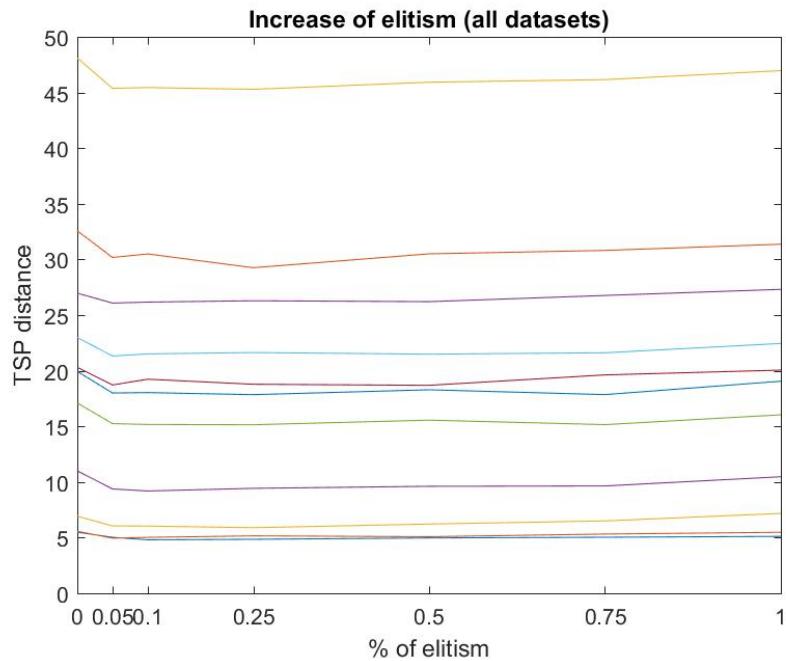
4.13.5 Modified-General-Individuals



4.13.6 Modified-General-Generations



4.13.7 Modified-General-Elitism



4.13.8 Modified-General-Crossover and mutation

