KU Leuven

Genetic Algorithms

TSP

Author:

Alejandro Rodríguez Salamanca: r0650814@student.kuleuven.be
Fernando Collado Egea: r0650586@student.kuleuven.be

January 11, 2017

Contents

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 adjancency 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 construct 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:

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

[1] This 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:

(1 2 — 3 4 5 — 6 7 8)
(2 4 — 6 8 7 — 5 3 1)

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:

(8 7 — 3 4 5 — 1 2 6)
(4 5 — 6 8 7 — 1 2 3)

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 **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 **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 analyize 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 refering 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 adecuate 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 it's 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 benefitial, 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 litle 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, alghouth 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 conclussion.

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 choosen 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.

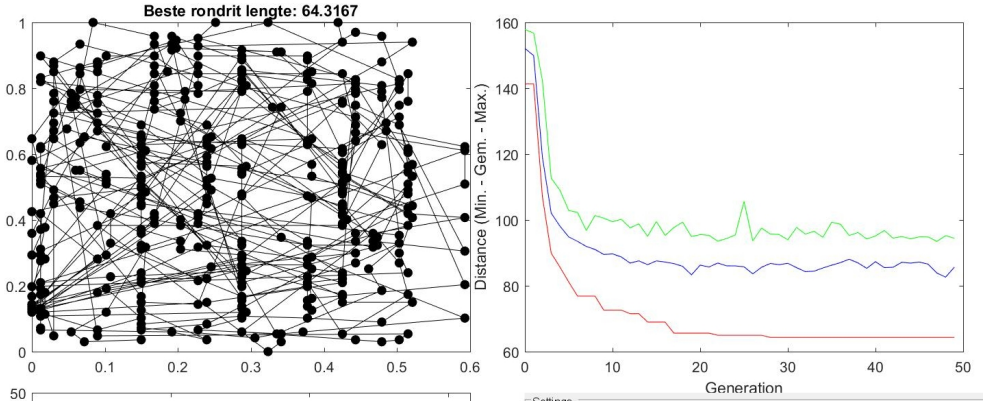
2.2 Specific tests

The specific tests are separeted 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

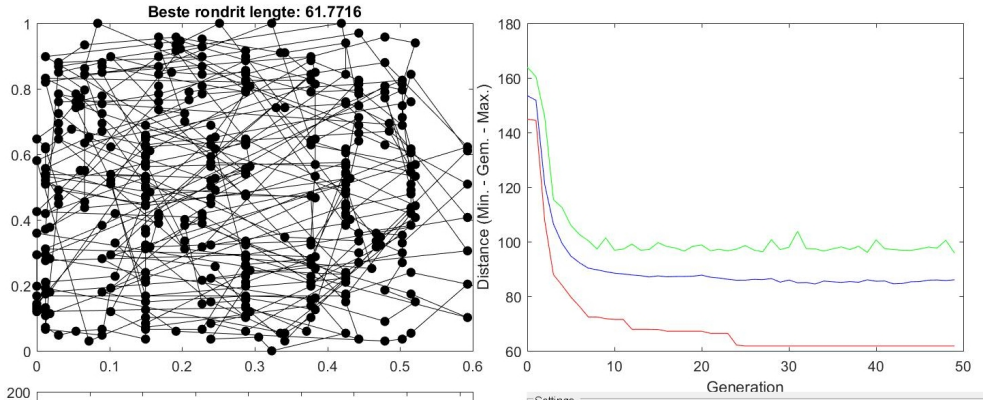
The base values are the same as with the general case. [Number of individuals, generations, percentage of elitism, of crossover and mutation] = [50,50,5%,95%, 5%]

The result obtained is



The test was executed in 12.66 sec

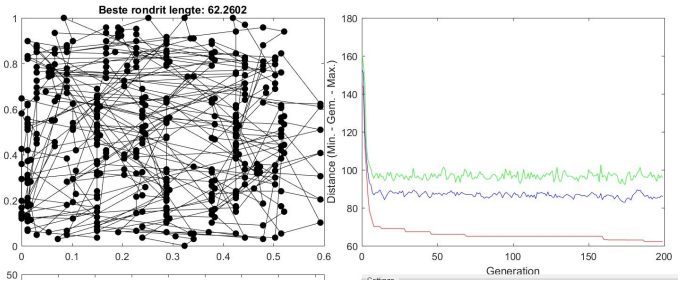
200 was the number of individuals that was deemed to be apropiate. The test, keeping all other values, but changing just the number of individuals is:
[Nind, gens, elitism, crossover, mutation] =[200,50,5%,95%,5%]



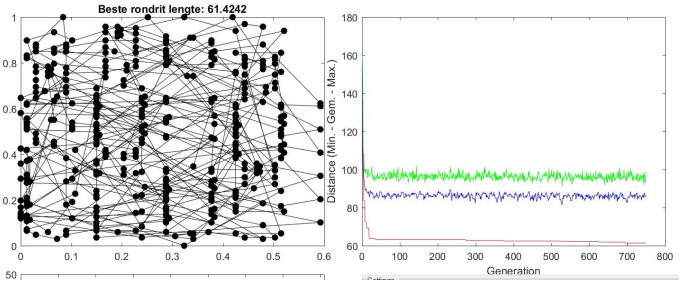
We can see that the minimum distance has decreased, from 64 to 61, not a big decrease, but significant enough. Increasing the number of individuals to 200 is then to be considered a good measure. Timewise, the test was executed in 31.61 seconds. More than twice the time for the first, which, given the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

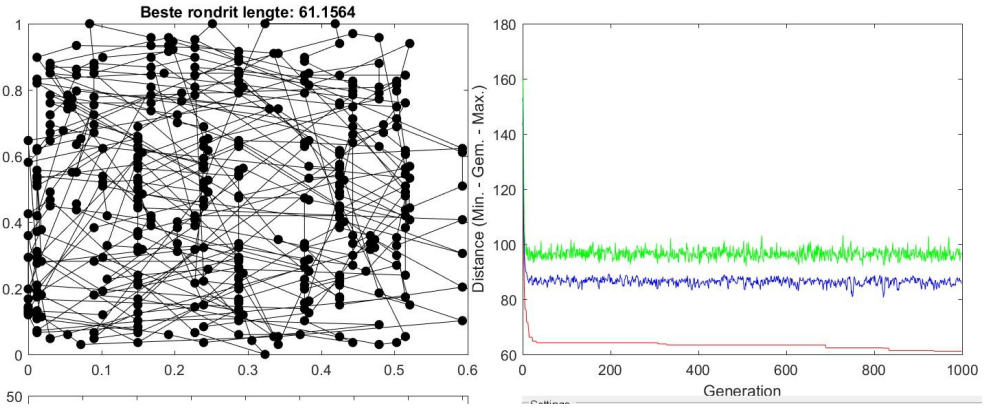
[50,200,5%,95%,5%]



[50,750,50,5%,95%,5%]

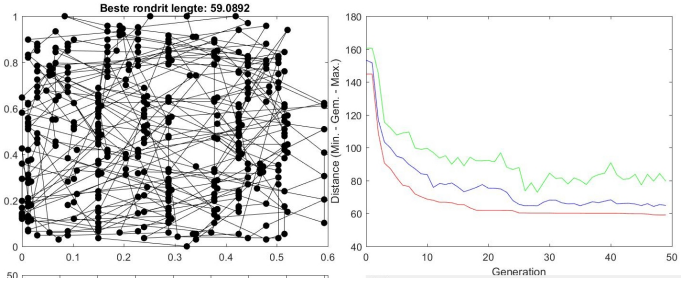


[50,1000,5%,95%,5%]

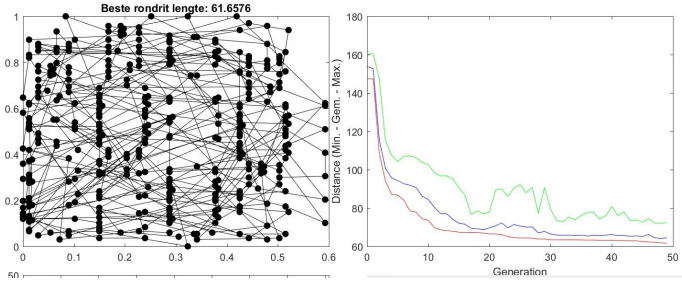


We can observe that the result is slightly better, with each increment of generations, but the cost is excesive, when it comes to time. The times for the tests are 46.01, 171.67, and 233.85 seconds, which means, 4, 13.5 and 18.5 times more than the base case.
As the elitism parameter was abundantly clear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

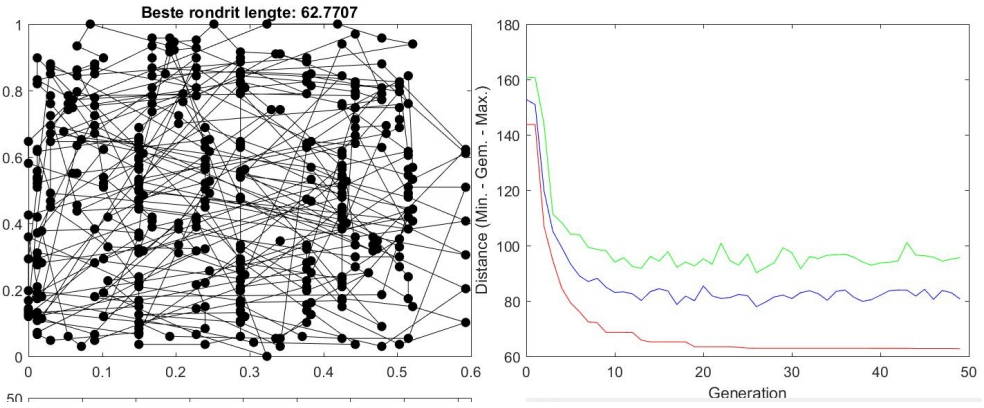
[50,50,5%,50%,50%]



[50,50,5,5%,80%,20%]



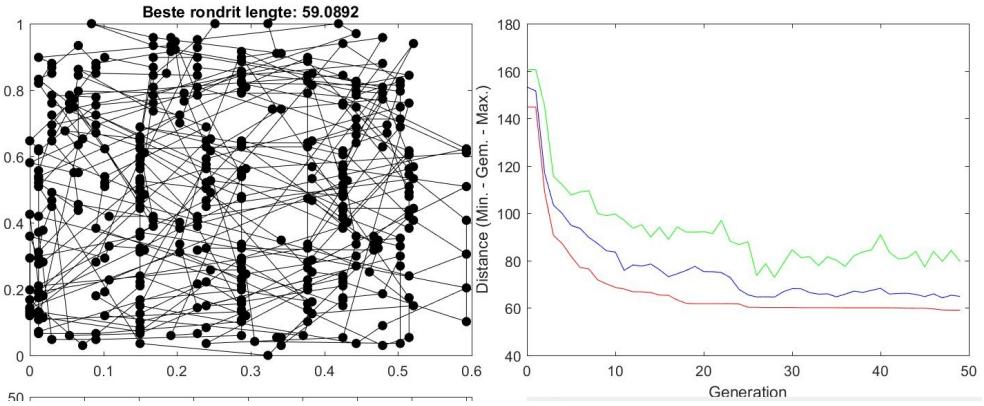
[50,50,5%,80%,20%]



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.

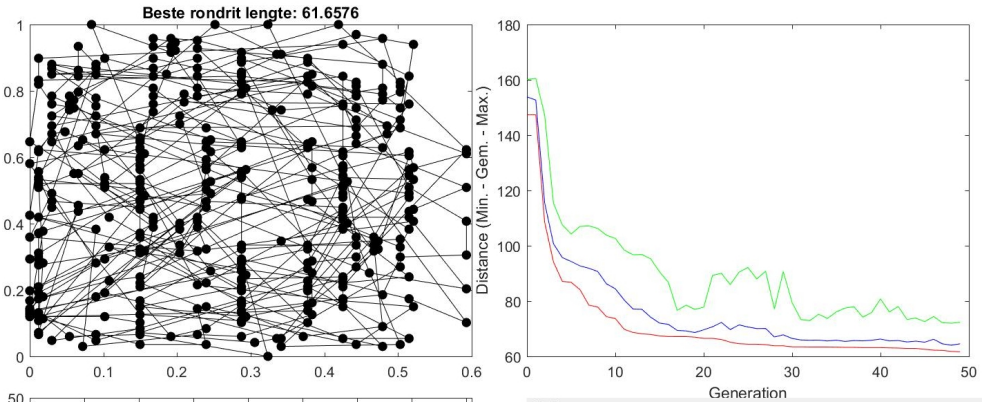
Finally, the combination of the good results is what made them get outstanding results. In order to avoid bias, the first combination is just modifying the number of individuals, and the number of generations

[200,200,5%,95%,5%]



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 choosen to be 200 because we deemed it good enough, and while costly, not as costly as higher quantities.

[200,200,5%,50%,50%]

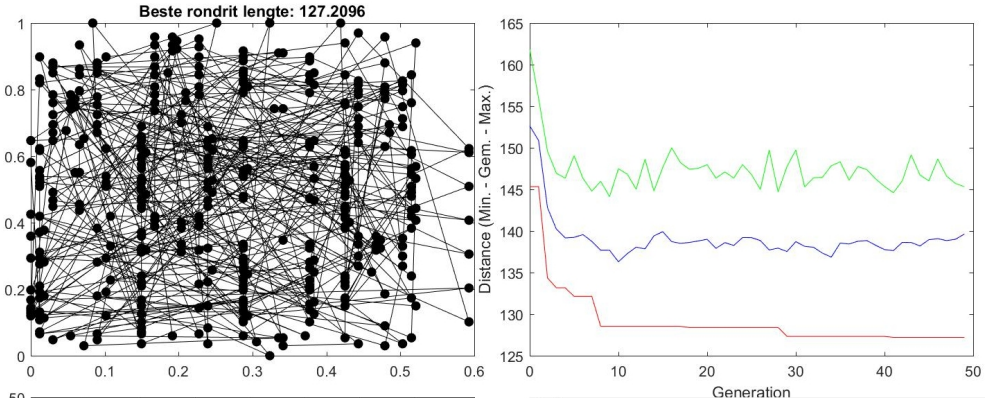


And the final result, when combining every single improvement, is the best of all. The decrease of distance is 30.8%, which could be said to be a great decrease. The time spent on this test was 76.74 seconds, around 6.1 times more than the original, although high, we consider that it is not sufficiently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

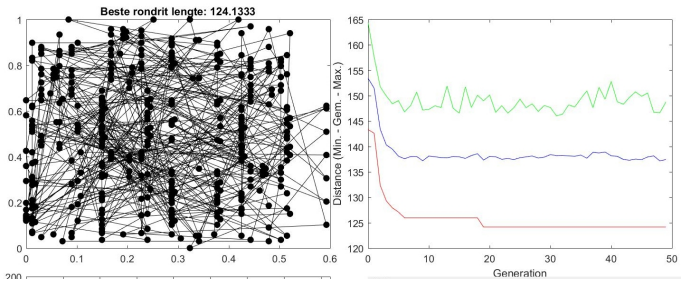
[50,50,5%,95%,5%]



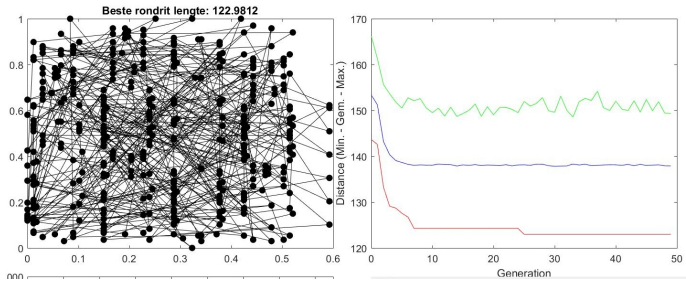
The first noticeable thing that we can observe here is that the distance is far higher than the adjacency representation, although looking at the time, it is better, since it takes only 6.88 seconds.

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.

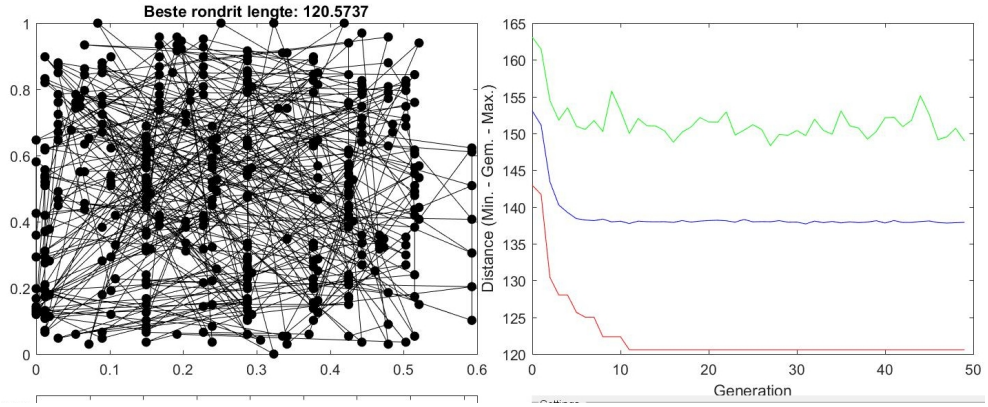
[200,50,5%,95%,5%]



[1000,50,50%,95%,5%]



[1250,50,5%,95%,5%]

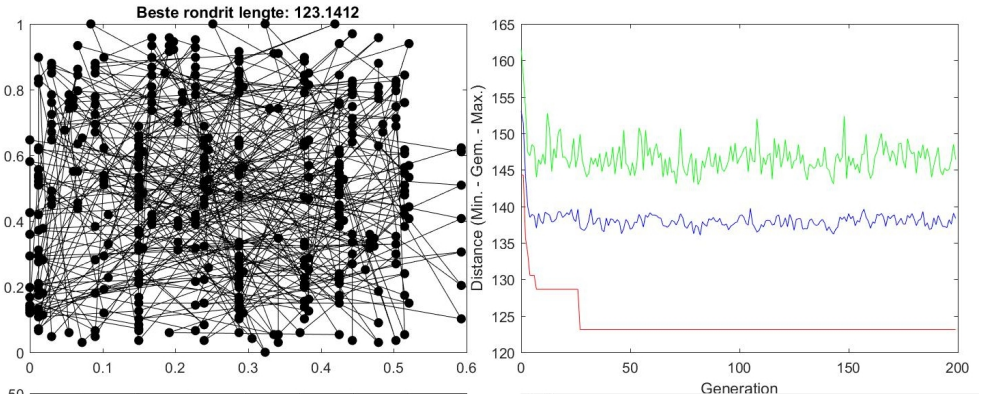


Although the change is not big, there is improvement, with each increase of individuals. And, what's more, as it is individuals, and not generations that we are increasnig now, there is higher cost, but not as high. The times for the tests are 9.08, 29.44, and 37.99 seconds,

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.

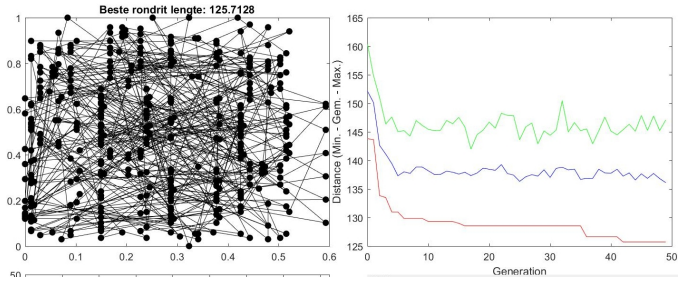
[50,200,5%,95%,5%]



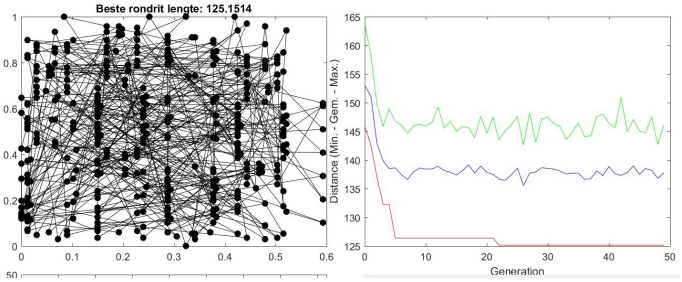
We can observe a slight improvement against the base case, 4 units of distance. Timewise, it is much more demanding that increasing individuals, since the test took 23.73 seconds to end, 3.5 times more than the base case. Any higher number of generations will only hinder the process by making it tediously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

[50,200,5%,10%,90%]



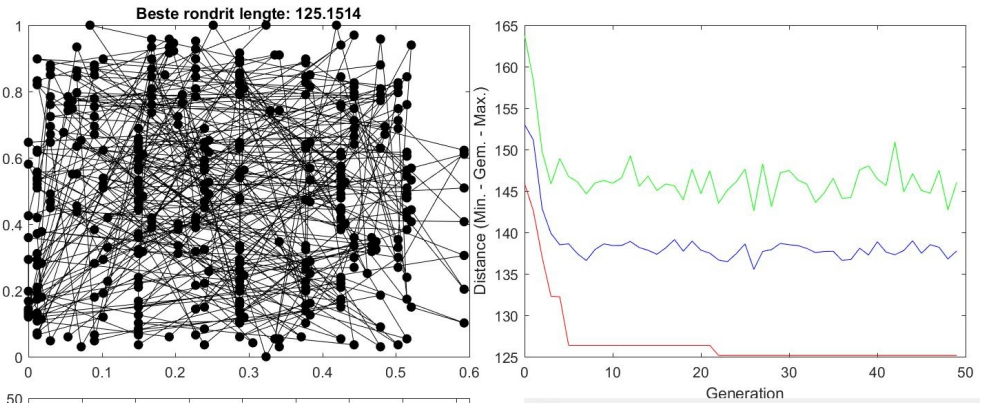
[50,200,5%,50%,50%]



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 represention, 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%. Obviosly, as the number of indviduals 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

[1250,200,5%,50%,50%]



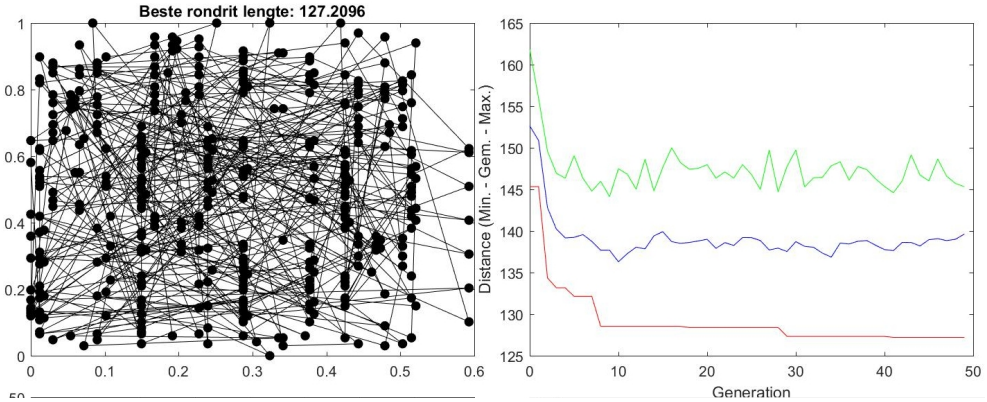
Unlike the previous combined test, the big expected improvement is nowhere to be seen. Going from a length of 127 to 120 is just a mere 5.2% of decrease. Furthermore, the combined increase of individuals and generations, given that the former is very high, makes the test to take extremly long compared 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 it is typically recommended [1].
This method consists of holding, in a round robin tournament way, competitions, and keeping the μ better individuals. The code for this implementation can be found in the appendix, in 3.12, `select_rr`, where the implementation for the round robin tournament is, and also, some minor modifications to `tspgui`, and `run_ga` were necessary, in order to make it work with both elitism and selection, depending on the value passed as parameter. For the test, `test.m` was modified, and now includes an option to perform the optional tests.

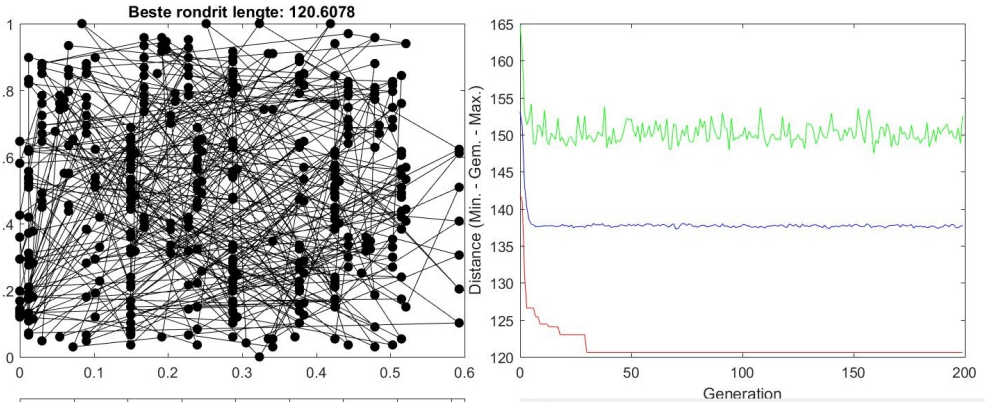
The tests made for this implementation, as it uses the crossover we implemented, we decided to test against the base case of said implementation.
As a reminder, here is the result for the base case

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



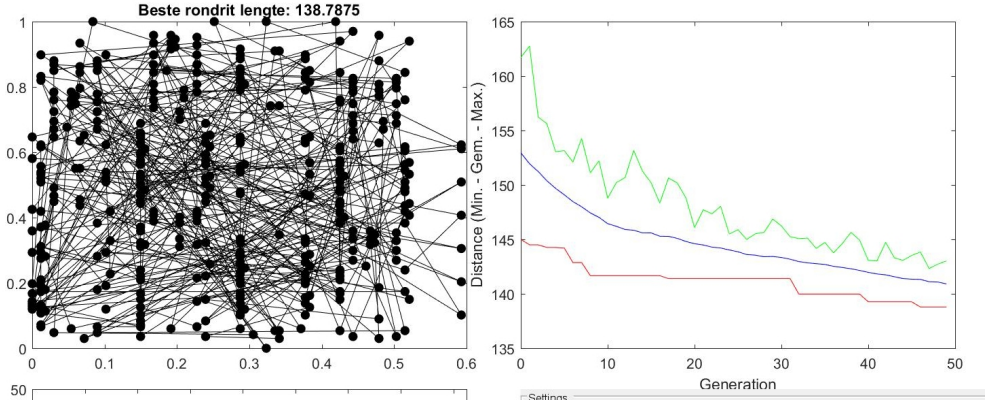
And, the best result obtained after modifying it, with the "best" parameters, according to our general tests

[1250,200,5%,50%,50%]



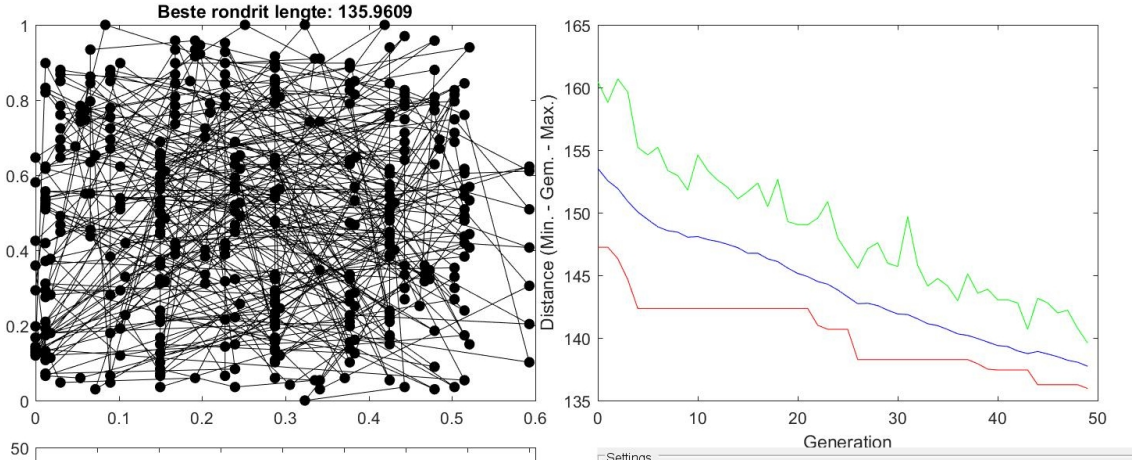
As can be seen, the new selection method started with a worse result, having the same paramters as the base case, only with the % of elitism changed to the number of selected individuals, 10

[50,50,10 selected,95%,5%]

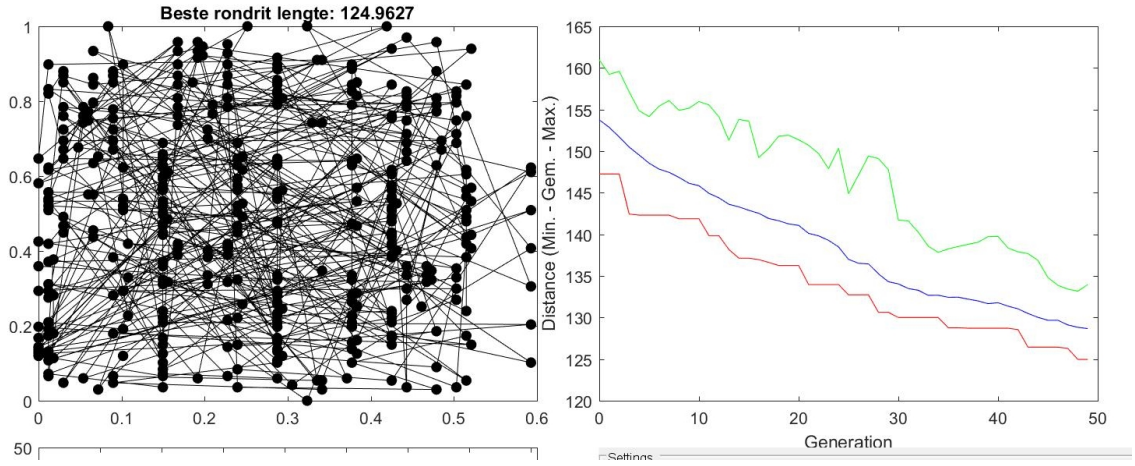


The only parameter that we can test for is the number of selected individuals from the tournament, and we did so, with 10 (the previous graph), 15,30, 45 and 50

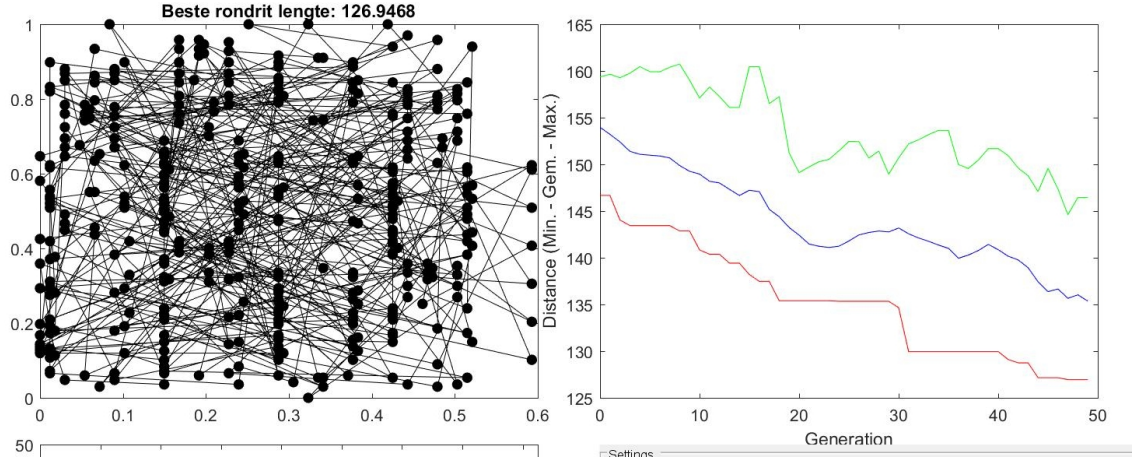
[50,50,15 selected,95%,5%]



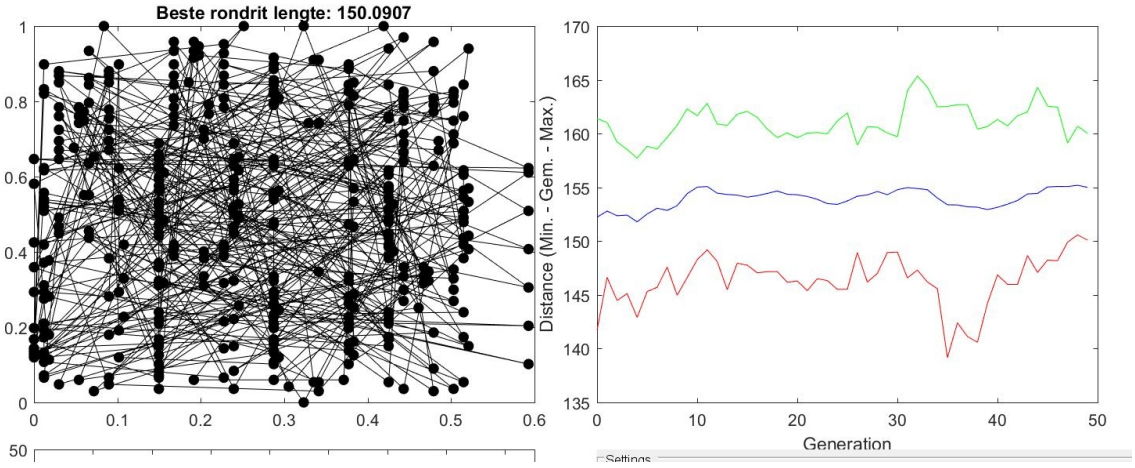
[50,50,30 selected,95%,5%]



[50,50,45 selected,95%,5%]



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



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 tunning 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

```
% tsp_ImprovePopulation.m
% Author: Mike Matton
%
% This function improves a tsp population by removing local loops from
% each individual.
%
% Syntax: improvedPopulation = tsp_ImprovePopulation(popsize, ncities, pop, improve, dists)
%
% Input parameters:
%   popsize           - The population size
%   ncities           - the number of cities
%   pop               - the current population (adjacency representation)
%   improve           - Improve the population (0 = no improvement, <>0 = improvement)
%   dists             - distance matrix with distances between the cities
%
% Output parameter:
%   improvedPopulation  - the new population after loop removal (if improve
%                         <> 0, else the unchanged population).

function newpop = tsp_ImprovePopulation(popsize, ncities, pop, improve,dists)

if (improve)
    for i=1:popsize

      result = improve_path(ncities, pop(i,:),dists);

      pop(i,:) = path2adj(result);

    end
end

newpop = pop;
```

4.2 run_ga.m

```
function run_ga(x, y, NIND, MAXGEN, NVAR, SELECTION,
   STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP, ah1,
   ah2, ah3)
% usage: run_ga(x, y,
%                NIND, MAXGEN, NVAR,
%                SELECTION, STOP_PERCENTAGE,
%                PR_CROSS, PR_MUT, CROSSOVER,
%                ah1, ah2, ah3)
%
%
% x, y: coordinates of the cities
% NIND: number of individuals
% MAXGEN: maximal number of generations

%  MODIFIED  %%%%%%%%%%
% SELECTION: if elitism: percentage of elite population, else,
%            it's mu, quantity of individuals to be selected
%  MODIFIED  %%%%%%%%%%


% STOP_PERCENTAGE: percentage of equal fitness (stop criterium)
% PR_CROSS: probability for crossover
% PR_MUT: probability for mutation
% CROSSOVER: the crossover operator
% calculate distance matrix between each pair of cities
% ah1, ah2, ah3: axes handles to visualise tsp
{NIND MAXGEN NVAR SELECTION STOP_PERCENTAGE PR_CROSS PR_MUT
   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);
```

```
for row=1:NIND
  %Chrom(row,:)=path2adj(randperm(NVAR));
    Chrom(row,:)=randperm(NVAR);
end
gen=0;
% number of individuals of equal fitness needed to stop
stopN=ceil(STOP_PERCENTAGE*NIND);
% evaluate initial population
ObjV = tspfun(Chrom,Dist);
best=zeros(1,MAXGEN);
% generational loop
while gen<MAXGEN
    sObjV=sort(ObjV);
    best(gen+1)=min(ObjV);
  minimum=best(gen+1);
    mean_fits(gen+1)=mean(ObjV);
    worst(gen+1)=max(ObjV);
    for t=1:size(ObjV,1)
        if (ObjV(t)==minimum)
            break;
        end
    end

    %visualizeTSP(x,y,adj2path(Chrom(t,:)), minimum, ah1,
       gen, best, mean_fits, worst, ah2, ObjV, NIND, ah3);
    visualizeTSP(x,y,Chrom(t,:), minimum, ah1, gen, best,
       mean_fits, worst, ah2, ObjV, NIND, ah3);

    if (sObjV(stopN)-sObjV(1) <= 1e-15)
        break;
    end
  %assign fitness values to entire population
  FitnV=ranking(ObjV);
  %select individuals for breeding
    if (SELECTION <=1)
        SelCh=select('sus', Chrom, FitnV, SELECTION);
    else
        SelCh=select_rr('sus', Chrom, FitnV, SELECTION);
    end
    %recombine individuals (crossover)
    SelCh = recombin(CROSSOVER,SelCh,PR_CROSS);
    %SelCh=mutateTSP('inversion',SelCh,PR_MUT);
    SelCh=mutateTSP('insertion',SelCh,PR_MUT); % <-- line
       changed, now insertion mutation is used
    %evaluate offspring, call objective function
```

```
            ObjVSel = tspfun(SelCh,Dist);
              %reinsert offspring into population
            [Chrom, ObjV]=reins(Chrom,SelCh,1,1,ObjV,ObjVSel);

              Chrom = tsp_ImprovePopulation(NIND, NVAR, Chrom,
                 LOCALLOOP, Dist);
            %increment generation counter
            gen=gen+1;

        end
        toc;
        minimum
end
```

4.3 insertion.m

```
% low level function for TSP mutation
% Representation is an integer specifying which encoding is used
% 1 : adjacency representation
% 2 : path representation
%

function NewChrom = insertion(OldChrom)

    NewChrom = OldChrom;
    % select two positions in the tour
    rndi = zeros(1,2);
    while rndi(1) == rndi(2)
        rndi=rand_int(1,2,[1 size(NewChrom,2)]);
    end
    rndi = sort(rndi);

    % get the value of the first random position
    temp = NewChrom(rndi(1));
    % insert this value in the second random position
    NewChrom = insertAt(NewChrom, temp, rndi(2));
    % remove the first random position
    NewChrom(rndi(1)) = [];
    % End of function
end

function arrOut = insertAt(arr,val,index)
    if index == numel(arr)+1
        arrOut = [arr val];
```

```
    else
        arrOut = [arr(1:index-1) val arr(index:end)];
    end
end
```

4.4 order_crossover.m

```
% Syntax:  NewChrom = order_crossover(OldChrom, XOVR)
%
% Input parameters:
%    OldChrom  - Matrix containing the chromosomes of the old
%                population. Each line corresponds to one individual
%                (in any form, not necessarily real values).
%    XOVR      - Probability of recombination occurring between pairs
%                of individuals.
%
% Output parameter:
%    NewChrom  - Matrix containing the chromosomes of the population
%                after mating, ready to be mutated and/or evaluated,
%                in the same format as OldChrom.
%

function NewChrom = order_crossover(OldChrom, XOVR)

if nargin < 2, XOVR = NaN; end
[rows,~]=size(OldChrom);

    maxrows=rows;
    if rem(rows,2)~=0
      maxrows=maxrows-1;
    end

    for row=1:2:maxrows

     % crossover of the two chromosomes
     % results in 2 offsprings
   if rand<XOVR       % recombine with a given probability
        MatrixChrom = order_low_level([OldChrom(row,:);OldChrom(
            row+1,:)]);
     NewChrom(row,:) = MatrixChrom(1, :);
```

```
33      NewChrom(row+1,:) = MatrixChrom(2, :);
34    else
35      NewChrom(row,:) = OldChrom(row,:);
36      NewChrom(row+1,:) = OldChrom(row+1,:);
37    end
38     end

40     if rem(rows,2) ~= 0
41       NewChrom(rows,:)=OldChrom(rows,:);
42     end

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:
            end_index);
19      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 -
                1);
50      end
51  % end function
```

4.6 tspgui.m

```
1       CROSSOVER = 'order_crossover';
2       crossover = uicontrol(ph,'Style','popupmenu', 'String',{'
            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
```

```
% ObjVal is a vector with the fitness values for each candidate tour
%   (=each row of Phen)
%

function ObjVal = tspfun(Phen, Dist)
    % the objective function works with adjacency representation. In this
    % version, path representation is used, so the fitness function should
    % be adapted. Now, the phenotype is converted to adjacency
    % representation first, and then, the Objective Value is computed as it
    % was computed in the original version.
    adj = zeros(size(Phen));
    for row=1:size(Phen)
        adj(row,:) = path2adj(Phen(row,:));
    end

    ObjVal = Dist(adj(:,1), 1);
  for t=2:size(adj,2)
      ObjVal=ObjVal + Dist(adj(:,t), t);
  end

% End of function
```

4.8 mutateTSP.m

```
% MUTATETSP.M       (MUTATion for TSP high-level function)
%
% This function takes a matrix OldChrom containing the
% representation of the individuals in the current population,
% mutates the individuals and returns the resulting population.
%
% Syntax:  NewChrom = mutate(MUT_F, OldChrom, MutOpt)
%
% Input parameter:
%    MUT_F     - String containing the name of the mutation function
%    OldChrom  - Matrix containing the chromosomes of the old
%                population. Each line corresponds to one individual.
%    MutOpt    - mutation rate
% Output parameter:
```

```
%    NewChrom  - Matrix containing the chromosomes of the
    population
%                after mutation in the same format as OldChrom.


function NewChrom = mutateTSP(MUT_F, OldChrom, MutOpt)

% Check parameter consistency
    if nargin < 2,  error('Not enough input parameters'); end

[rows,~]=size(OldChrom);
NewChrom=OldChrom;

for r=1:rows
  if rand<MutOpt
    NewChrom(r,:) = feval(MUT_F, OldChrom(r,:));
  end
end

% End of function
```

4.9 run_ga_test.m

```
function minimum=run_ga_test(modality, x, y, NIND, MAXGEN, NVAR,
    SELECTION, ...
     STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP)
% usage: run_ga(x, y,
%                NIND, MAXGEN, NVAR,
%                ELITIST, STOP_PERCENTAGE,
%                PR_CROSS, PR_MUT, CROSSOVER,
%                ah1, ah2, ah3)
%
%
% x, y: coordinates of the cities
% NIND: number of individuals
% MAXGEN: maximal number of generations
% ELITIST: percentage of elite population
% STOP_PERCENTAGE: percentage of equal fitness (stop criterium)
% PR_CROSS: probability for crossover
% PR_MUT: probability for mutation
% CROSSOVER: the crossover operator
% calculate distance matrix between each pair of cities
% ah1, ah2, ah3: axes handles to visualise tsp
```

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


        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);
        for row=1:NIND
            if (modality==1)
                Chrom(row,:)=randperm(NVAR);
            else
                Chrom(row,:)=path2adj(randperm(NVAR));
            end

        end
        gen=0;
        % number of individuals of equal fitness needed to stop
        stopN=ceil(STOP_PERCENTAGE*NIND);
        % evaluate initial population
        ObjV = tspfun(Chrom, Dist);
        best=zeros(1,MAXGEN);
        % generational loop
        while gen<MAXGEN
            sObjV=sort(ObjV);
            best(gen+1)=min(ObjV);
          minimum=best(gen+1);
            mean_fits(gen+1)=mean(ObjV);
            worst(gen+1)=max(ObjV);
            for t=1:size(ObjV,1)
                if (ObjV(t)==minimum)
                    break;
                end
            end

```

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

4.10 tspgui_test.m

```
1  function tspgui_test(CROSSOVER, NIND, MAXGEN,SELECTION,PR_CROSS,
       PR_MUT, ...
2      LOCALLOOP, testName)
3
4
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
if(SELECTION<=1)
    GGAP=1-SELECTION;   % Generation gap
    sliderTxt='% elite';
    maxSlider=100;
    minSlider=0;
    val=round(SELECTION*100);
else
    sliderTxt='# indviduals';
    maxSlider=NIND;
    minSlider=10;
    val=round(SELECTION);
end

STOP_PERCENTAGE=.95;    % percentage of equal fitness individuals for stopping
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% read an existing population
% 1 -- to use the input file specified by the filename
% 0 -- click the cities yourself, which will be saved in the file called
%USE_FILE=0;
%FILENAME='data/cities.xy';
%if (USE_FILE==0)
%    % get the input cities
%     fg1 = figure(1);clf;
%    %subplot(2,2,2);
%     axis([0 1 0 1]);
%     title(NVAR);
%     hold on;
%     x=zeros(NVAR,1);y=zeros(NVAR,1);
%     for v=1:NVAR
%         [xi,yi]=ginput(1);
%         x(v)=xi;
%         y(v)=yi;
%         plot(xi,yi, 'ko','MarkerFaceColor','Black');
%         title(NVAR-v);
%     end
%     hold off;
%     set(fg1, 'Visible', 'off');
```

```
%     dlmwrite(FILENAME,[x y],'\t');
%else
%     XY=dlmread(FILENAME,'\t');
%     x=XY(:,1);
%     y=XY(:,2);
%end

% load the data sets from the benchmark list
benchmarks = dir('TSPBenchmark/');
datasets=cell( size(benchmarks,1)-2,1);
for i=1:size(datasets,1)
     datasets{i} = benchmarks(i+2).name;
end

% start with first dataset
data = load(['TSPBenchmark/bcl380.tsp']);
x=data(:,1)/max([data(:,1);data(:,2)]);
y=data(:,2)/max([data(:,1);data(:,2)]);
NVAR=size(data,1);

% initialise the user interface
fh = figure('Visible','off','Name','TSP Tool','Position'
    ,[0,0,1024,768]);
ah1 = axes('Parent',fh,'Position',[.1 .55 .4 .4]);
plot(x,y,'ko')
ah2 = axes('Parent',fh,'Position',[.55 .55 .4 .4]);
axes(ah2);
xlabel('Generation');
ylabel('Distance (Min. - Gem. - Max.)');
ah3 = axes('Parent',fh,'Position',[.1 .1 .4 .4]);
axes(ah3);
title('Histogram');
xlabel('Distance');
ylabel('Number');

ph = uipanel('Parent',fh,'Title','Settings','Position',[.55 .05
    .45 .45]);
datasetpopuptxt = uicontrol(ph,'Style','text','String','Dataset','
    Position',[0 260 130 20]);
datasetpopup = uicontrol(ph,'Style','popupmenu','String',datasets,
    'Value',1,'Position',[130 260 130 20],'Callback',
    @datasetpopup_Callback);
llooppopuptxt = uicontrol(ph,'Style','text','String','Loop
    Detection','Position',[260 260 130 20]);
```

```
82 llooppopup = uicontrol(ph,'Style','popupmenu','String',{'off','on'
   },'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',sliderTxt,'
   Position',[0 80 130 20]);
99 elitslider = uicontrol(ph,'Style','slider','Max',maxSlider,'Min',
   minSlider,'Value',val,'Sliderstep',[0.01 0.05],'Position',[130
   80 150 20],'Callback',@elitslider_Callback);
100 elitsliderv = uicontrol(ph,'Style','text','String',val,'Position'
   ,[280 80 50 20]);
```

```
crossover = uicontrol(ph,'Style','popupmenu', 'String',{'
    order_crossover'}, 'Value',1,'Position',[10 50 130 20],'
    Callback',@crossover_Callback);
%inputbutton = uicontrol(ph,'Style','pushbutton','String','Input
    ','Position',[55 10 70 30],'Callback',@inputbutton_Callback);
runbutton = uicontrol(ph,'Style','pushbutton','String','START','
    Position',[0 10 50 30],'Callback',@runbutton_Callback);

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

%%%%%%%%%%%%
%Added for the tests
runbutton_Callback;
saveas(fh, testName, 'jpg');
close(fh);
%%%%%%%%%%%%

    function datasetpopup_Callback(hObject,~)
        dataset_value = get(hObject,'Value');
        dataset = datasets{dataset_value};
        % load the dataset
        data = load(['datasets/' dataset]);
        x=data(:,1)/max([data(:,1);data(:,2)]);y=data(:,2)/max([
            data(:,1);data(:,2)]);
        %x=data(:,1);y=data(:,2);
        NVAR=size(data,1);
        set(ncitiessliderv,'String',size(data,1));
        axes(ah1);
        plot(x,y,'ko')
    end
    function llooppopup_Callback(hObject,~)
        lloop_value = get(hObject,'Value');
        if lloop_value==1
            LOCALLOOP = 0;
        else
            LOCALLOOP = 1;
        end
    end
    function ncitiesslider_Callback(hObject,~)
        fslider_value = get(hObject,'Value');
        slider_value = round(fslider_value);
        set(hObject,'Value',slider_value);
        set(ncitiessliderv,'String',slider_value);
        NVAR = round(slider_value);
    end
```

```
141     function nindslider_Callback(hObject,~)
142         fslider_value = get(hObject,'Value');
143         slider_value = round(fslider_value);
144         set(hObject,'Value',slider_value);
145         set(nindsliderv,'String',slider_value);
146         NIND = round(slider_value);
147     end
148     function genslider_Callback(hObject,~)
149         fslider_value = get(hObject,'Value');
150         slider_value = round(fslider_value);
151         set(hObject,'Value',slider_value);
152         set(gensliderv,'String',slider_value);
153         MAXGEN = round(slider_value);
154     end
155     function mutslider_Callback(hObject,~)
156         fslider_value = get(hObject,'Value');
157         slider_value = round(fslider_value);
158         set(hObject,'Value',slider_value);
159         set(mutsliderv,'String',slider_value);
160         PR_MUT = round(slider_value)/100;
161     end
162     function crossslider_Callback(hObject,~)
163         fslider_value = get(hObject,'Value');
164         slider_value = round(fslider_value);
165         set(hObject,'Value',slider_value);
166         set(crossslidery,'String',slider_value);
167         PR_CROSS = round(slider_value)/100;
168     end
169     function elitslider_Callback(hObject,~)
170         fslider_value = get(hObject,'Value');
171         slider_value = round(fslider_value);
172         set(hObject,'Value',slider_value);
173         set(elitsliderv,'String',slider_value);
174         if(SELECTION<=1)
175             SELECTION = round(slider_value)/100;
176             GGAP = 1-SELECTION;
177         else
178             SELECTION = round(slider_value);
179         end
180     end
181     function crossover_Callback(hObject,~)
182         crossover_value = get(hObject,'Value');
183         crossovers = get(hObject,'String');
184         CROSSOVER = crossovers(crossover_value);
185         CROSSOVER = CROSSOVER{1};
```

```
186     end
187     function runbutton_Callback(~,~)
188         %set(ncitiesslider, 'Visible','off');
189         set(nindslider,'Visible','off');
190         set(genslider,'Visible','off');
191         set(mutslider,'Visible','off');
192         set(crossslider,'Visible','off');
193         set(elitslider,'Visible','off');
194         run_ga(x, y, NIND, MAXGEN, NVAR, SELECTION,
               STOP_PERCENTAGE, PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP
               , ah1, ah2, ah3);
195         end_run();
196     end
197     function inputbutton_Callback(~,~)
198         [x,y] = input_cities(NVAR);
199         axes(ah1);
200         plot(x,y,'ko')
201     end
202     function end_run()
203         %set(ncitiesslider,'Visible','on');
204         set(nindslider,'Visible','on');
205         set(genslider,'Visible','on');
206         set(mutslider,'Visible','on');
207         set(crossslider,'Visible','on');
208         set(elitslider,'Visible','on');
209     end
210 end
```

4.11 test.m

```
1  %%%%
2  %{
3  Modifications:
4
5  run_ga_test --> Now returns a value, minimun, 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
```

```
test --> There are 2 sets of tests. First, specific tests, limited in
number, made with a purpose. Those tests are run with tspgui_test, using
the dataset from the benchmark bcl380, and saving the figure.
The second set is the generalistic tests, which check the 4 different
components to be changed (aka: number of individuals, number of
generations, elitism, and percentage of crossover and mutation), with all
the datasets. Each test is made with a certain number of values,
repeated *reps* (variable in the code, currently set to 5) times, and the
mean obtained by that is the value we use for the plots.


%}
%%%%

%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 choosen

%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)

        %Depending on the representation (crossover choosen), the
           data
        % for the specific test is choosen
        if(strcmp(CROSSOVER,'order_crossover'))

            %
               %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            NIND=      [50,200,1000,1250,50,50,50,1250];
            MAXGEN=    [50,50,50,50,200,50,50,200];
            ELITIST=0.05;
            PR_CROSS=[.95,.95,.95,.95,.95,.1,.5,.5,.5];
            PR_MUT=    [.05,.05,.05,.05,.05,.9,.5,.5];
            LOCALLOOP=1;
            %
               %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


        else

            %
               %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

            NIND=      [50,200,50,50,50,50,50,50,200,200];
            MAXGEN=    [50,50,200,750,1000,50,50,50,200,200];
            ELITIST=0.05;
            PR_CROSS=[.95,.95,.95,.95,.95,.5,.20,.80,.95,.5];
            PR_MUT=    [.05,.05,.05,.05,.05,0.5,.8,.2,.05,.5];
            LOCALLOOP=1;
            %
               %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


        end

        %
           %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


```

```
        upbound = size(NIND);
        for i=1:upbound(2)
            tspgui_test(CROSSOVER, NIND(i), MAXGEN(i),ELITIST(i), ...
            PR_CROSS(i),PR_MUT(i),LOCALLOOP, ...
                strcat( 'tests/', CROSSOVER,'/','general_',num2str(i) ) );
        end

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

    %General tests
    if(n==2 || n==3)

        reps = 5;

        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
        NIND=[50,100,150,200,500,750,1000];
        MAXGEN=[50,100,150,200,500,750,1000];
        ELITIST=[0,0.05, 0.1, 0.25, 0.5, 0.75, 1] ;
        STOP_PERCENTAGE=.95;
        PR_CROSS=[0, 0.05, 0.1, 0.25, 0.5, 0.75, 1];
        PR_MUT=[1,0.95, 0.9, 0.75, 0.5, 0.25, 0];
        LOCALLOOP=1;
        %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

        datasetslist = dir('datasets/');
        datasets=cell( size(datasetslist,1)-2 ,1);

        upbound = size(NIND);
        modality = 0;

        if (strcmp(CROSSOVER,'order_crossover'))
            modality=1;
        end

        %possible values for each test, city configurations,# of different tests
        values = zeros(upbound(2),size(datasets,1),4);

        for j=1:upbound(2)
```

```
124             for i=1:size(datasets,1)
125
126                 datasets{i} = datasetslist(i+2).name;
127                 data = load(['datasets/' datasets{i}]);
128                 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),
                            MAXGEN(1),...
137                                         NVAR, ELITIST(1), ...
138                                         STOP_PERCENTAGE, PR_CROSS(1),
                                            PR_MUT(1),...
139                                         CROSSOVER,LOCALLOOP );
140                 end
141                 values(j,i,1) = median(temp)   ;
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),
                                        PR_MUT(1),...
150                                   CROSSOVER,LOCALLOOP );
151                 end
152                 values(j,i,2) = median(temp)   ;
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),
                            MAXGEN(1), ...
158                                 NVAR, ELITIST(j), ...
159                                 STOP_PERCENTAGE, PR_CROSS(1),
                                        PR_MUT(1),...
160                                   CROSSOVER,LOCALLOOP );
161                 end
162                 values(j,i,3) = median(temp)   ;
163
```

```
164                 %Tests for % of crossover and mutation
165                 temp = zeros(reps,1);
166                 for n=1:reps
167                     temp(n)=run_ga_test(modality,x,y,NIND(1),
                           MAXGEN(1), ...
168                                 NVAR, ELITIST(1), ...
169                                 STOP_PERCENTAGE, PR_CROSS(j),
                                       PR_MUT(j),...
170                                   CROSSOVER,LOCALLOOP );
171                 end
172                 values(j,i,4) = median(temp)   ;
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'),
                'jpg');
188         close(nindF);
189
190         %Number of gen plot
191         genF = figure;
192         p = plot(NIND,values(:,:,2));
193         xlabel('# of Generations');
194         title('Increase of generations (all datasets)');
195         ylabel('TSP distance');
196         set(gca,'XTick', MAXGEN);
197         saveas(genF, strcat('tests/',CROSSOVER, '/numberGens'), '
               jpg');
198         close(genF);
199
200         %Elitism plot
201         elitF = figure;
202         p = plot(ELITIST,values(:,:,3));
203         xlabel('% of elitism');
204         title('Increase of elitism (all datasets)');
```

```
205         ylabel('TSP distance');
206         set(gca, 'XTick', ELITIST);
207         saveas(elitF, strcat('tests/',CROSSOVER, '/elitism'), 'jpg
                ');
208         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', '
                0.5|0.5'...
217               '0.75|0.25', '1|0'};
218         set(gca,'XLim',[1 size(PR_CROSS,2)],'XTick',1:size(
                PR_CROSS,2) ...
219           ,'XTickLabel',xlab)
220         saveas(porcF, strcat('tests/',CROSSOVER, '/crossMut'), '
                jpg');
221         close(porcF);
222 
223     end %if (n==2 || n==3)
224 end %end of function
225 
226 function perform_optest()
227 
228     %Testing optional
229     datasetslist = dir('datasets/');
230     datasets=cell( size(datasetslist,1)-2,1);
231     for i=1:size(datasets,1)
232     datasets{i} = datasetslist(i+2).name;
233     end
234     data = load(['datasets/' datasets{1}]);
235     x=data(:,1)/max([data(:,1);data(:,2)]);
236     y=data(:,2)/max([data(:,1);data(:,2)]);
237 
238     NVAR=size(data,1);
239 
240     NIND=50;     % Number of individuals
241     MAXGEN=50;     % Maximum no. of generations
242     SELECTION=[10,15,30,45,50];     % Number of indv to be selected
            after tournament
243     PR_CROSS=.95;       % probability of crossover
244     PR_MUT=.05;         % probability of mutation
```

```
245     LOCALLOOP=0;        % local loop removal
246     CROSSOVER = 'order_crossover';  % default crossover operator
247 
248     upbound = size(SELECTION);
249     for i=1:upbound(2)
250         tspgui_test(CROSSOVER, NIND, MAXGEN,SELECTION(i), ...
251         PR_CROSS,PR_MUT,LOCALLOOP, ...
252             strcat( 'tests/optional_',num2str(i) ) );
253     end
254 
255     %val = run_ga_test(1,x,y,NIND, MAXGEN, NVAR, SELECTION,
        STOP_PERCENTAGE, ...
256     %PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP);
257     %display(val);
258 end
```

4.12 select_rr.m

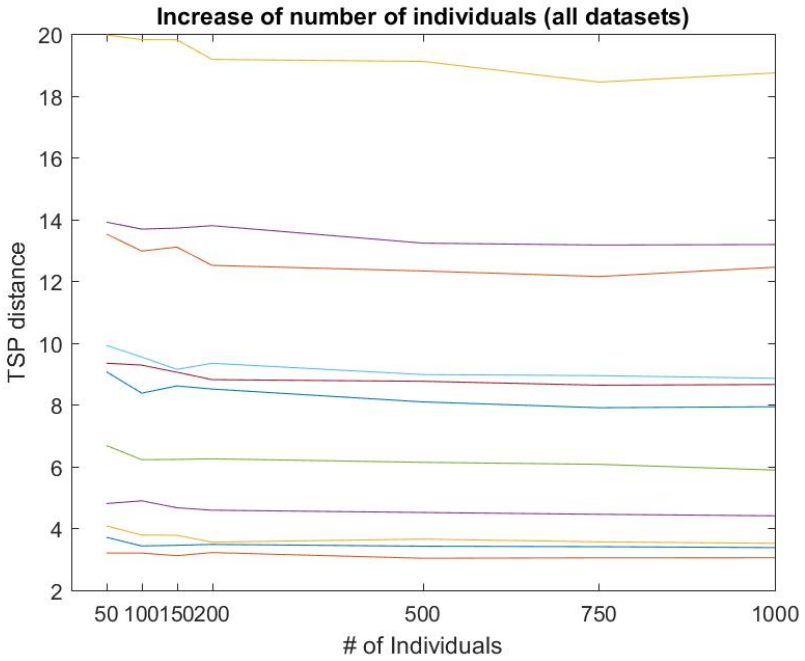
```
1  % Modified SELECT.M, now uses round robin tournament instead of
       elitism
2  
3  % Syntax:  SelCh = select_rr(SEL_F, Chrom, FitnV, GGAP)
4  %
5  % Input parameters:
6  %    SEL_F     - Name of the selection function
7  %    Chrom     - Matrix containing the individuals (parents) of
       the current
8  %                population. Each row corresponds to one
       individual.
9  %    FitnV     - Column vector containing the fitness values of
       the
10 %                individuals in the population.
11 %    GGAP      - (optional) Rate of individuals to be selected
12 %                if omitted 1.0 is assumed
13 %
14 % Output parameters:
15 %    SelCh     - Matrix containing the selected individuals.
16 
17 function SelCh = select_rr(SEL_F, Chrom, FitnV, mu)
18 
19     % Check parameter consistency
20     if nargin < 3, error('Not enough input parameter'); end
21 
22     % Identify the population size (Nind)
```

```
    [NindCh,~] = size(Chrom);
    [NindF,VarF] = size(FitnV);
    if NindCh ~= NindF, error('Chrom and FitnV disagree'); end
    if VarF ~= 1, error('FitnV must be a column vector'); end

    Nind = NindCh;

    if nargin < 4, mu = 10; end

    if nargin > 3
        if isempty(mu), mu = 10;
            elseif isnan(mu), mu = 10;
            elseif length(mu) ~= 1, error('Mu must be a scalar');
            elseif (mu < 9), error('Mu must be a scalar bigger
               than 9');
        end
    end

    %Select q random individuals
    q = zeros(10,1);
    temp = zeros(10,1);
    for i=1:10
        ind = randi(size(Chrom,1)) ;

        %Avoid repetition
        while(any(ind==temp))
            ind=randi(size(Chrom,1)) ;
        end

        temp(i) = ind;
        q(i) = FitnV(ind);
    end


    %Tournament variable, only index and amount of wins is stored
    tournament = zeros(size(Chrom,1),2);
    %Tournament-> every indv compites against q rivals
    for i=1:size(Chrom,1)
        tournament(i,1) = i;
        for j=1:size(q,1)
            if(FitnV(i) > q(j))
                tournament(i,2) = tournament(i,2) + 1;
            end
        end
    end
```

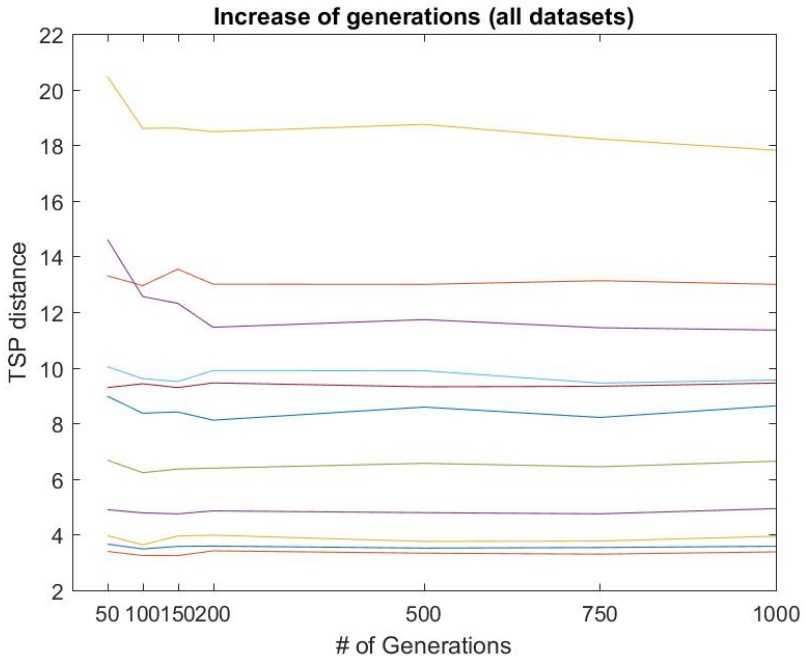
```
67
68      %Sort tournament by amount of wins
69      sortedTournament = sortrows(tournament,2);
70
71      %Indexes of the mu better individuals
72      indexes = sortedTournament(size( ...
73          sortedTournament,1)-mu+1:size(sortedTournament,1),1);
74
75
76      SelCh = [];
77      FitnVSub = FitnV(indexes);
78      ChrIx=feval(SEL_F, FitnVSub, mu);
79      SelCh=[SelCh; Chrom(ChrIx,:)];
80
81  % End of function
```

4.13 Graph

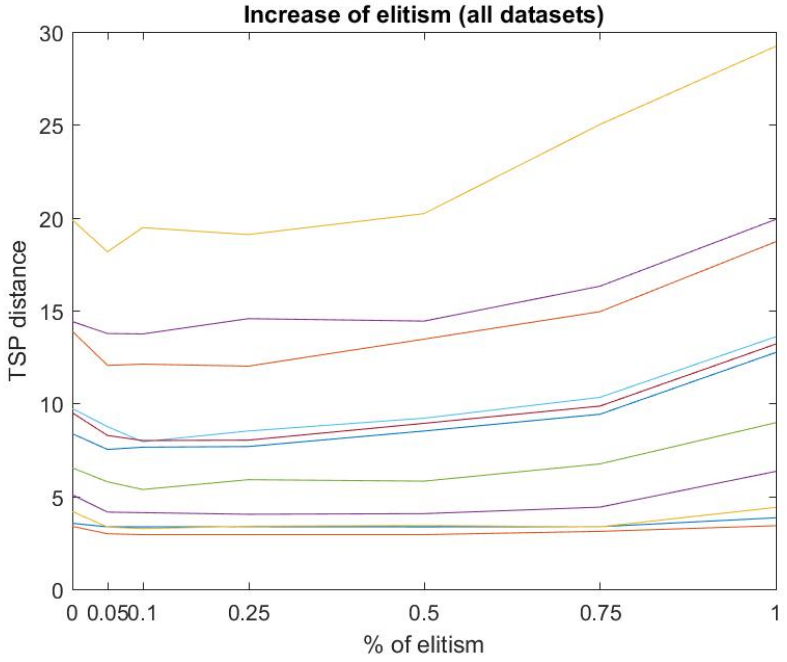
4.13.1 Original-General-Individuals



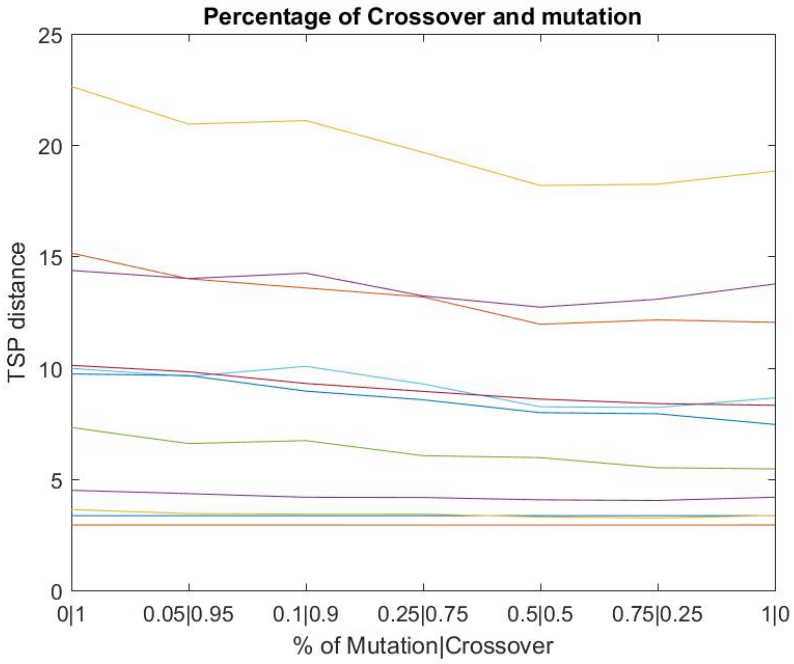
4.13.2 Original-General-Generations



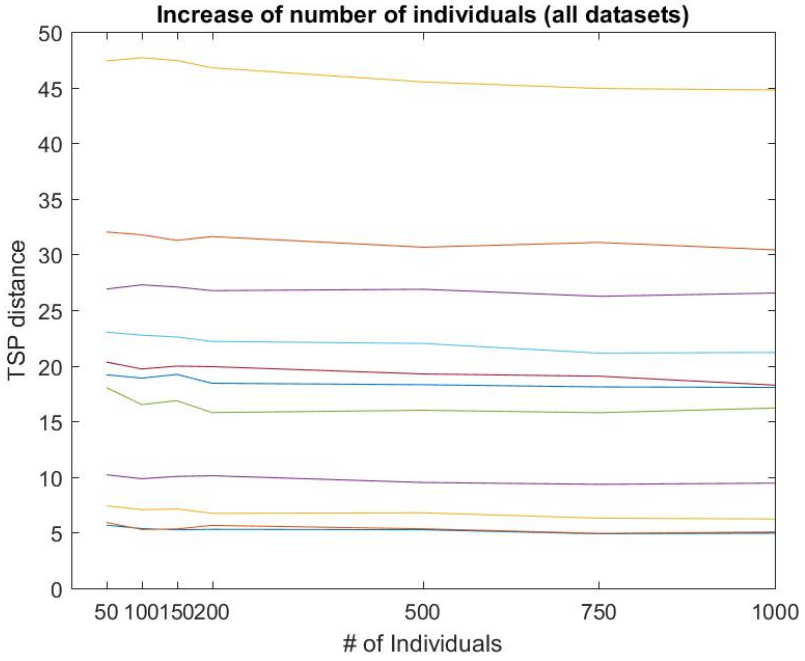
4.13.3 Original-General-Elitism



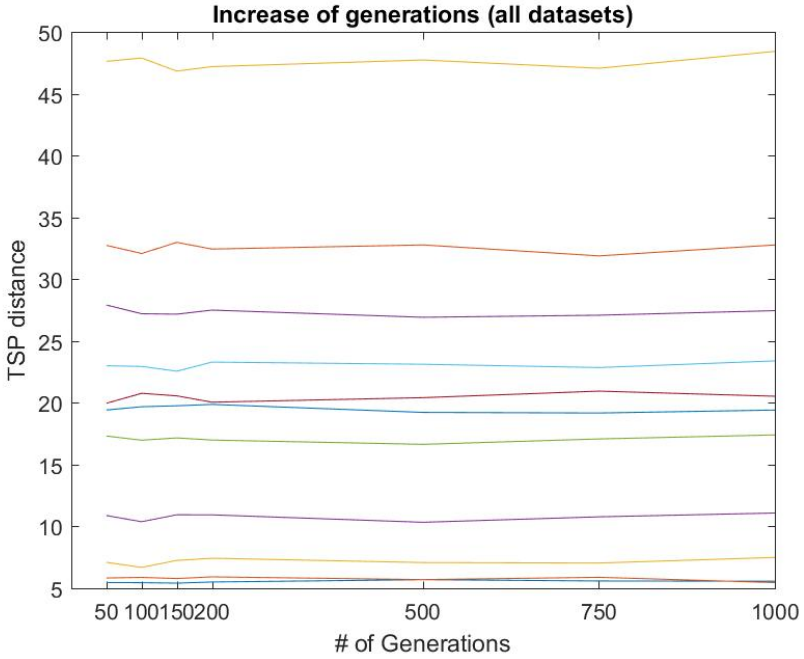
4.13.4 Original-General-Crossover and mutation



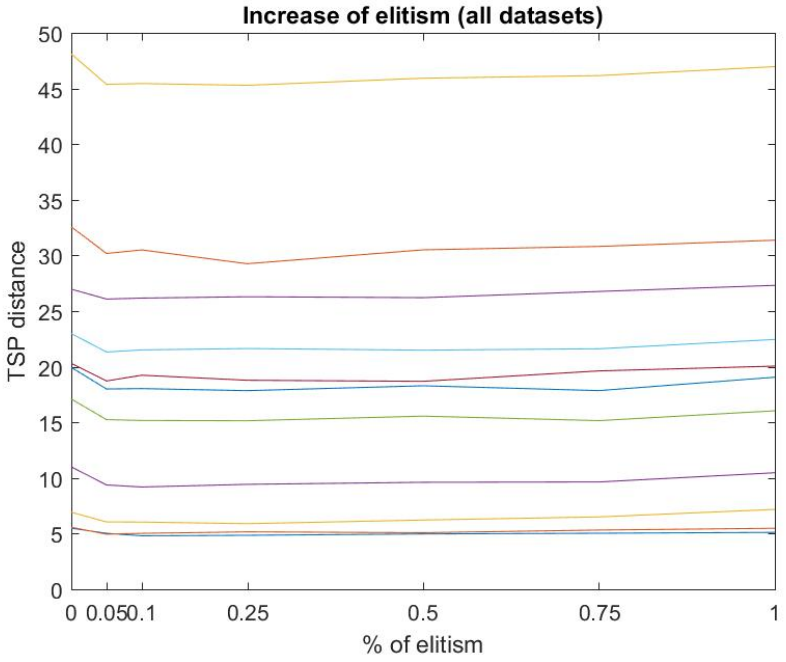
4.13.5 Modified-General-Individuals



4.13.6 Modified-General-Generations



4.13.7 Modified-General-Elitism



4.13.8 Modified-General-Crossover and mutation