KU Leuven

Genetic Algorithms

TSP

Author:

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

January 9, 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)
For the general tests, the default values are:

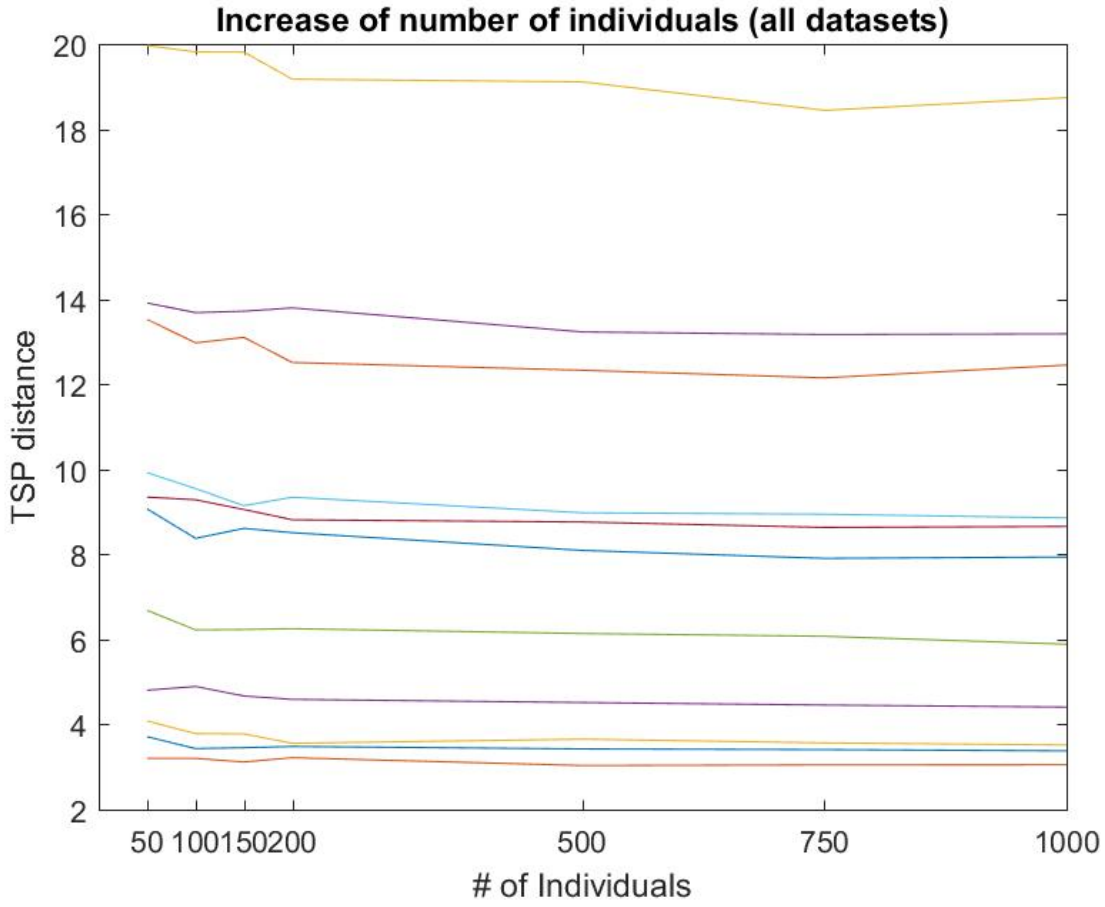
- Number of individuals - 50
- Number of generations - 50
- Elitism - 0.05
- Crossover - 0.95
- Mutation - 0.05
- Stop percentage condition - 0.95
- Detection of loops on

And, the different values for each modified parameter are

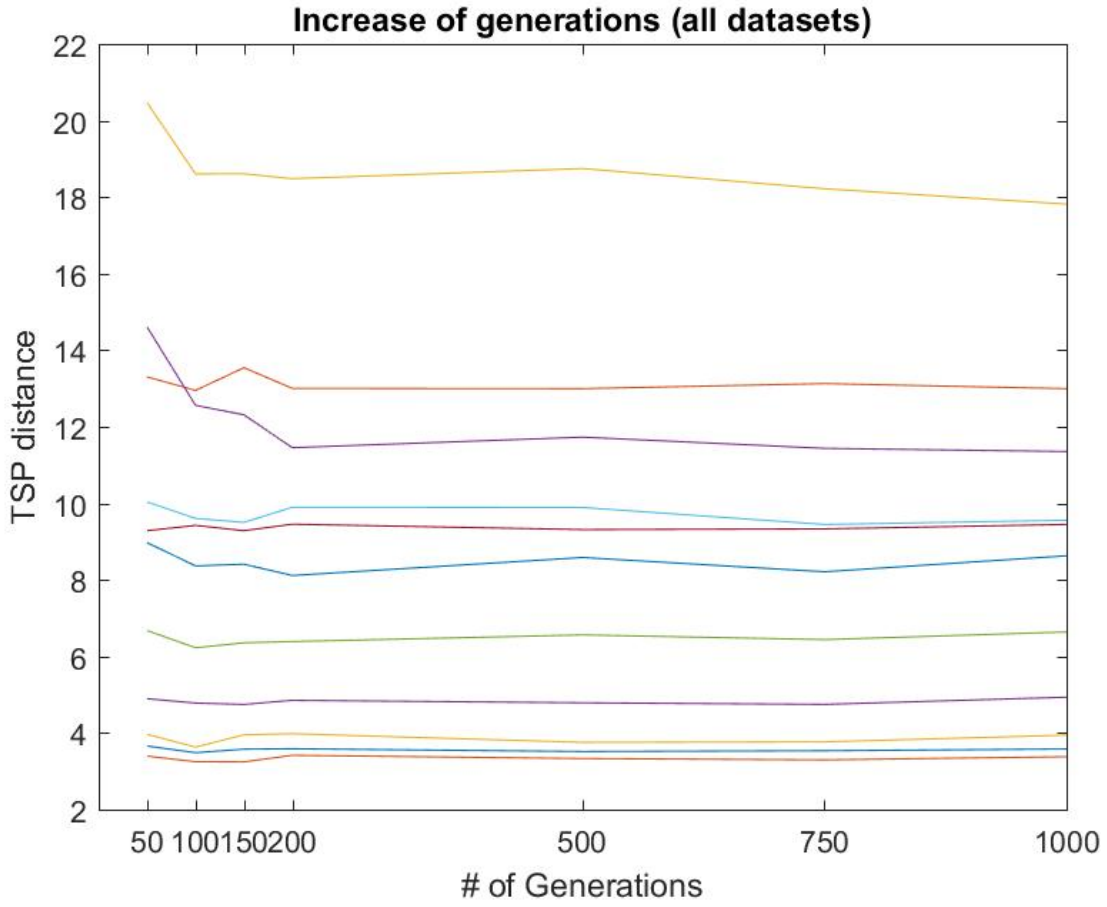
- Number of individuals - [50,100,150,200,500,750,1000]
- Number of individuals - [50,100,150,200,500,750,1000]
- Elitism percentage - [0,0.05,0.1,0.2,0.5,0.75,1]
- Crossover—Mutation balance - [1—0,0.95—0.05,0.9—0.1,0.75—0.25, 0.5—0.5,0.25—0.75,0—1]

2.1 Adjacency representation

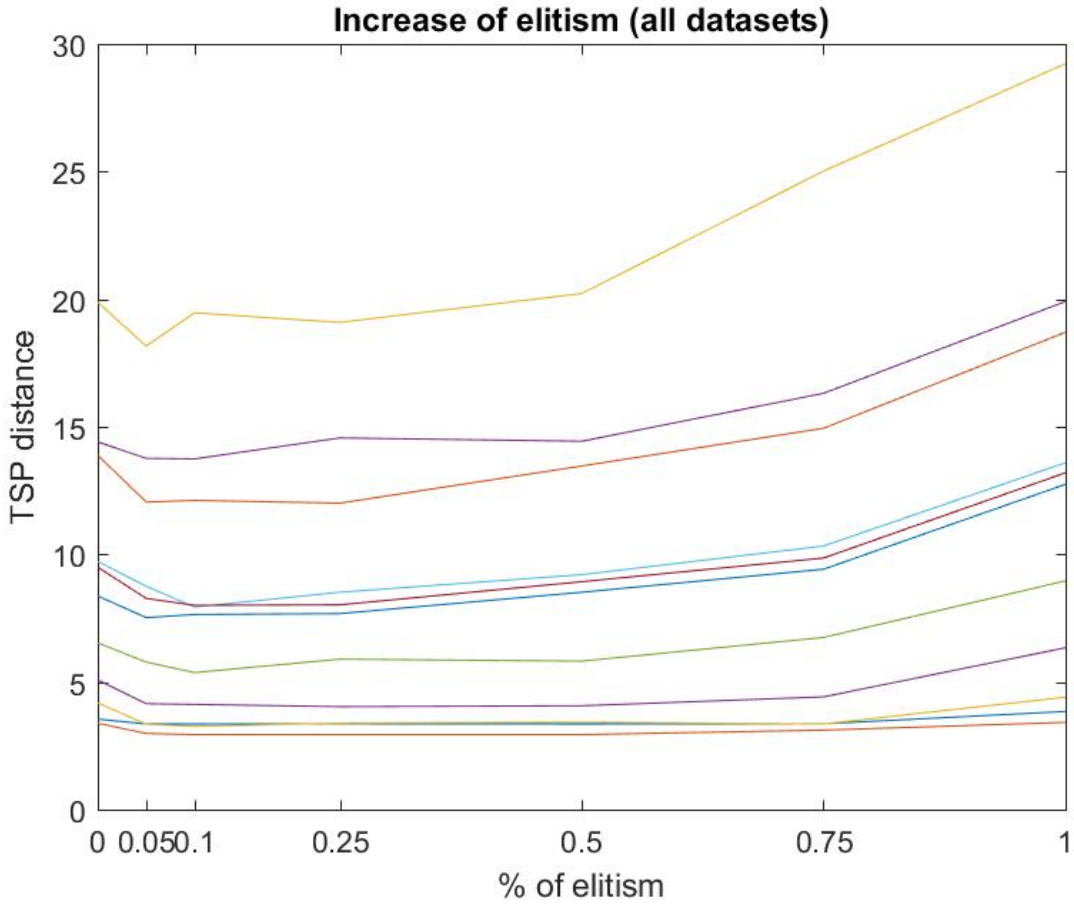
With the provided representation, adjacency representation, the results of the general tests are:



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. Almost all datasets start in global maximum, and only a couple in a local max. However, the mayority 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, as happens with the highest, and third highest dataset, thus 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.

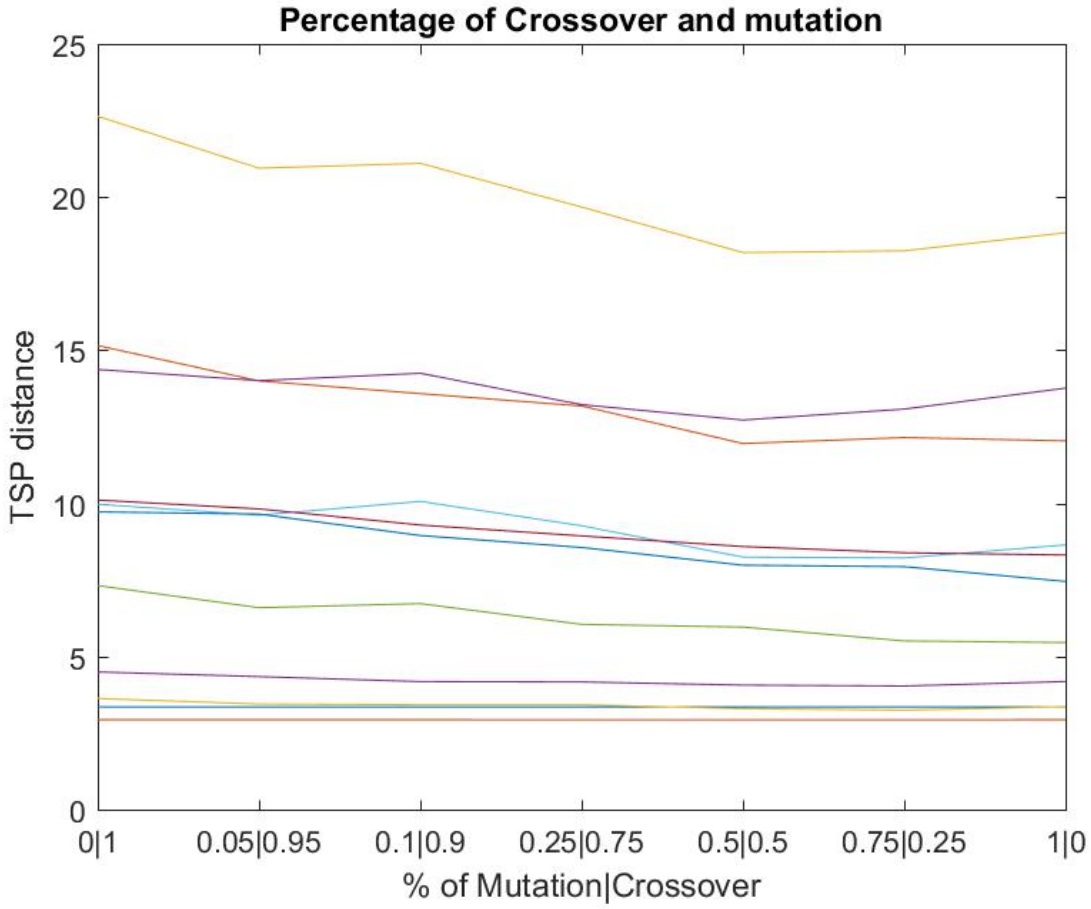


Once again, it is to be expected that a low quantity of generations will not yield good results. But that is not the only thing the number of individuals and generations have in common, since it seems like 200 is one of the best options for the number of generations. There are some differences, for instance, 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, the results are much more clear. 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.

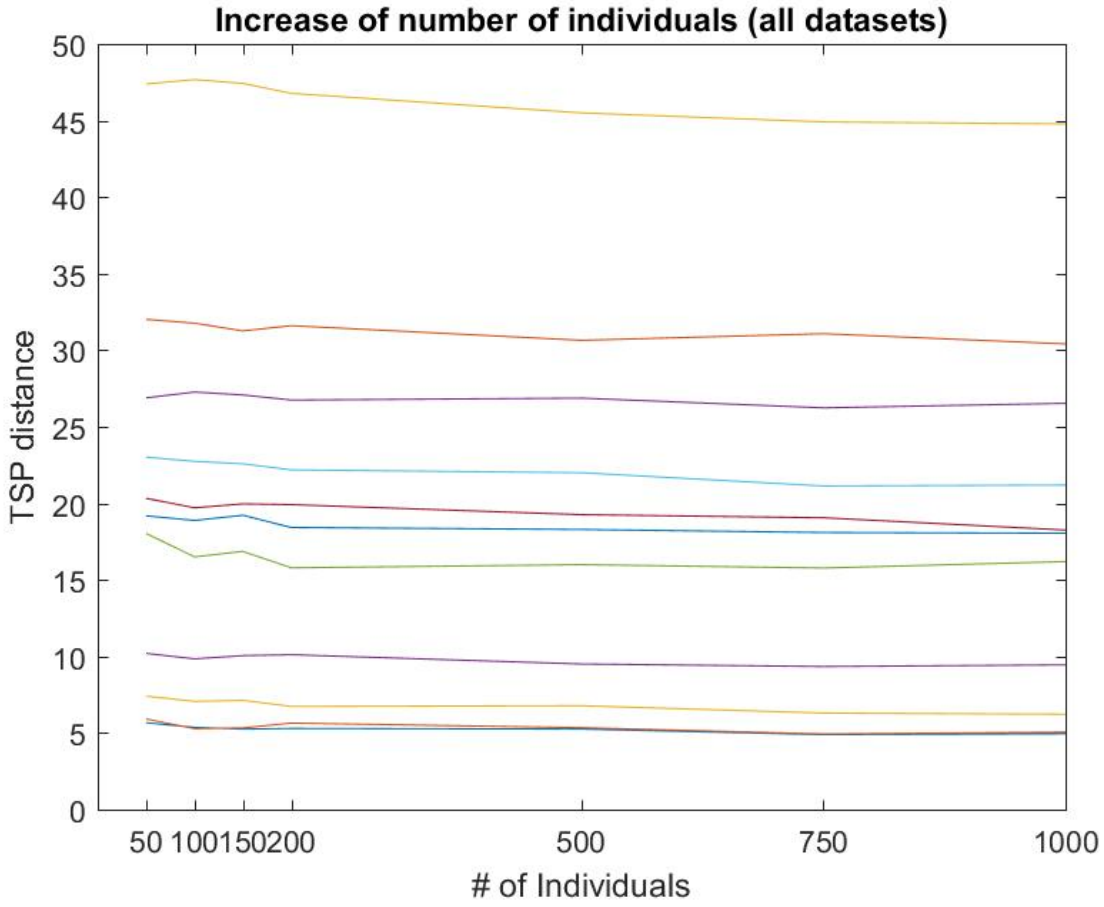


The test reflects perfectly the balance between exploitation and exploration, the overall result shows that the best performance comes when mutation has a value of 0.5, and crossover 0.5.

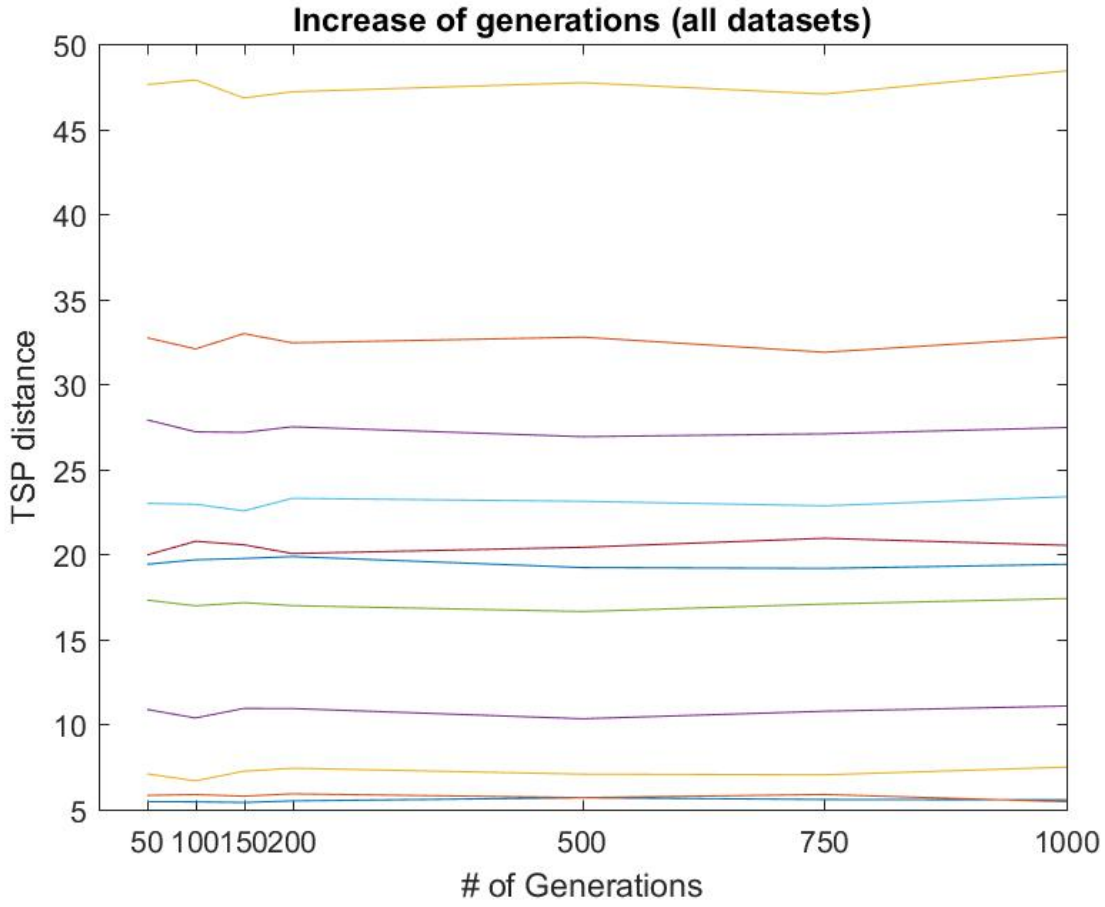
Any value of mutation higher than 0.5, and the results start to worsen, because there is too much exploitation, and too litle exploration. Any value of mutation lower than 0.5, and the results, most of the cases, are far worse. This leads to the conclusion that 0.5 is the candidate for the specific tests,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

2.2 Path representation

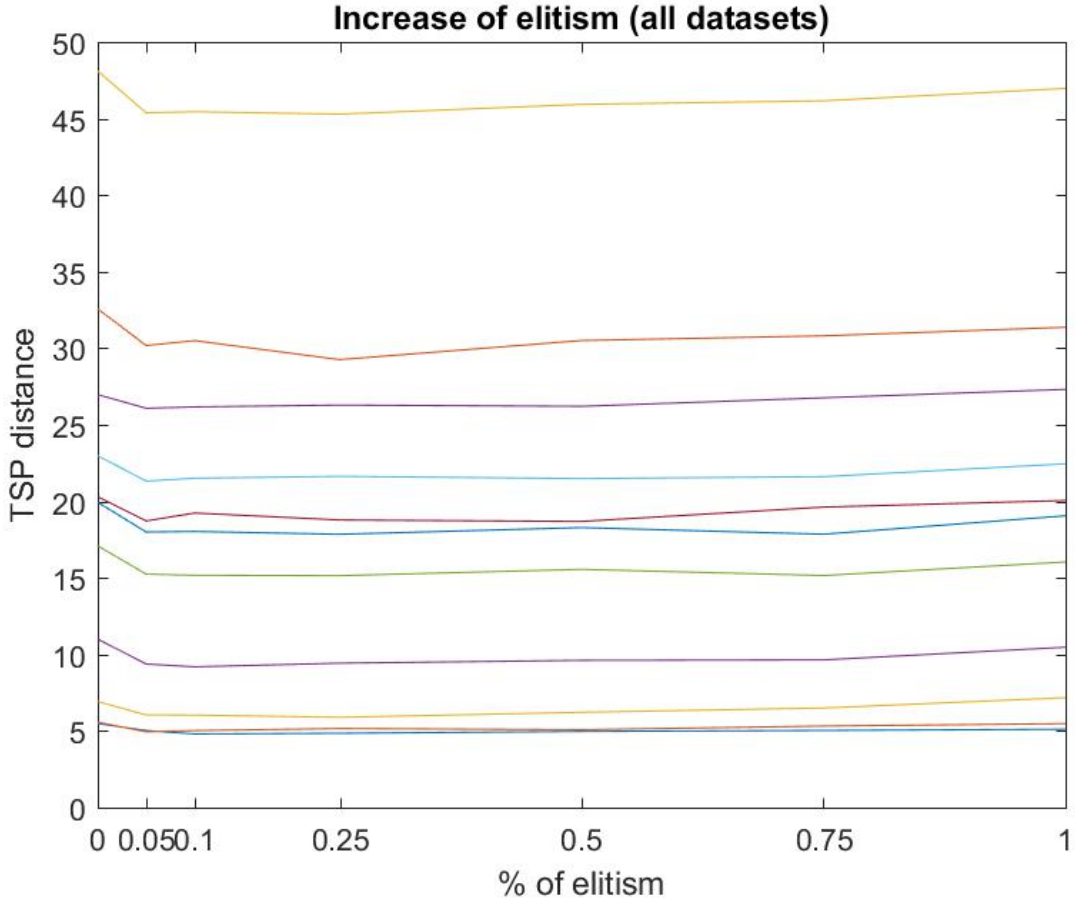
As explained in the implementation section, the representation we implemented is path representation. The results after executing the same tests as before are



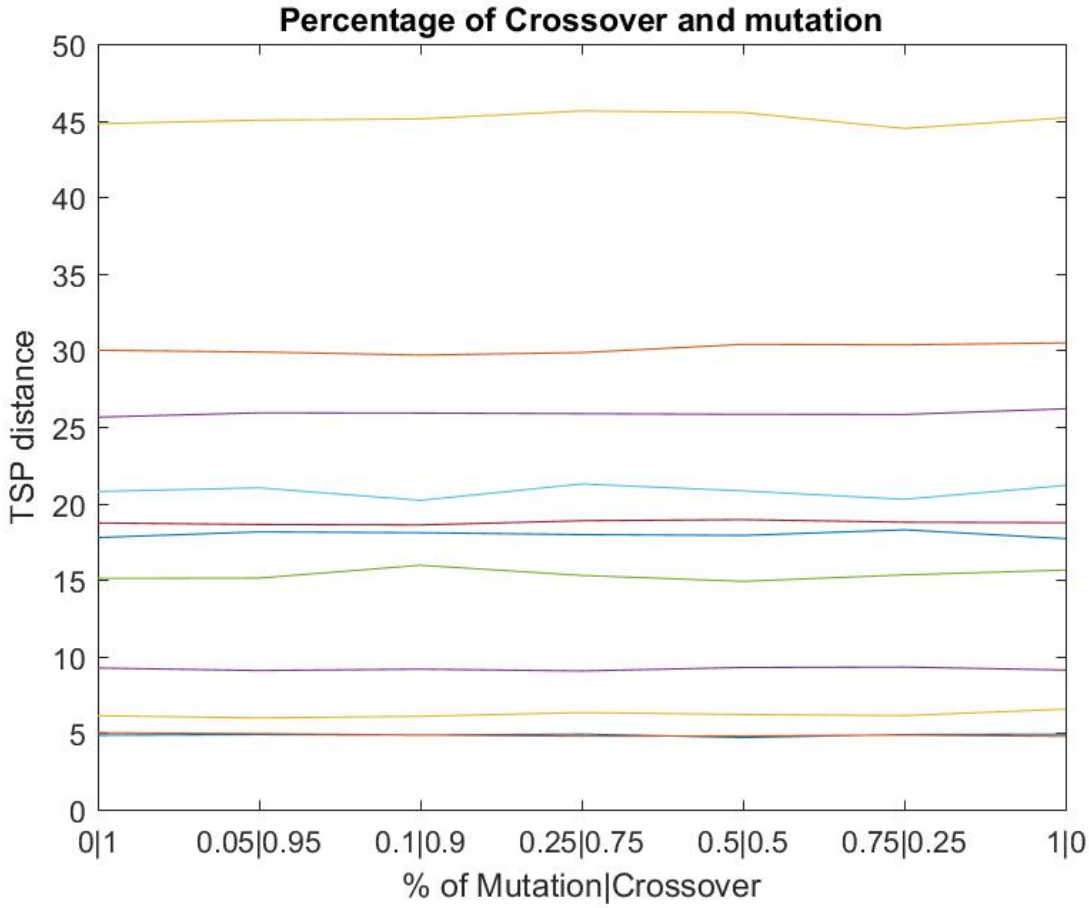
Similarly to adjacency representation, the result for the test of increasing the number of individuals has a generally located minimum local at 200, although for some cases, the distance becomes constant at 100. The only noticeable difference is that it is more stable at lower quantities of individuals, and 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.



Once again, as expected, there is a number of individuals from which the change in the result is null. That point, as can be observed, is 200 individuals, the same as the other representation. But then again, there are differences, two main ones, the first, the change from 50 to 100 individuals is not so apparent, and 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.



From what can be observed, 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



The result of this test is daunting. It was repeated, in case it was somehow erroneous, but the same graph was obtained. With our representation, there seems to be no effect whatsoever on the exploitation vs exploration dilemma. It does not matter, apparently, the percentage of mutation or crossover

2.3 Specific tests

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

3 Apendix

3.1 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;
```

3.2 run_ga.m

```
function run_ga(x, y, NIND, MAXGEN, NVAR, ELITIST, STOP_PERCENTAGE
    , PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP, ah1, ah2, ah3)
% 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
{NIND MAXGEN NVAR ELITIST STOP_PERCENTAGE PR_CROSS PR_MUT
   CROSSOVER LOCALLOOP};

        tic;
        GGAP = 1 - ELITIST;
        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);
```

```
42          best=zeros(1,MAXGEN);
43          % generational loop
44          while gen<MAXGEN
45              sObjV=sort(ObjV);
46              best(gen+1)=min(ObjV);
47            minimum=best(gen+1);
48              mean_fits(gen+1)=mean(ObjV);
49              worst(gen+1)=max(ObjV);
50              for t=1:size(ObjV,1)
51                  if (ObjV(t)==minimum)
52                      break;
53                  end
54              end
55
56              %visualizeTSP(x,y,adj2path(Chrom(t,:)), minimum, ah1,
                    gen, best, mean_fits, worst, ah2, ObjV, NIND, ah3);
57              visualizeTSP(x,y,Chrom(t,:), minimum, ah1, gen, best,
                    mean_fits, worst, ah2, ObjV, NIND, ah3);
58
59              if (sObjV(stopN)-sObjV(1) <= 1e-15)
60                  break;
61              end
62            %assign fitness values to entire population
63            FitnV=ranking(ObjV);
64            %select individuals for breeding
65            SelCh=select('sus', Chrom, FitnV, GGAP);
66            %recombine individuals (crossover)
67              SelCh = recombin(CROSSOVER,SelCh,PR_CROSS);
68              %SelCh=mutateTSP('inversion',SelCh,PR_MUT);
69              SelCh=mutateTSP('insertion',SelCh,PR_MUT); % <-- line
                    changed, now insertion mutation is used
70              %evaluate offspring, call objective function
71            ObjVSel = tspfun(SelCh,Dist);
72              %reinsert offspring into population
73            [Chrom, ObjV]=reins(Chrom,SelCh,1,1,ObjV,ObjVSel);
74
75              Chrom = tsp_ImprovePopulation(NIND, NVAR, Chrom,
                    LOCALLOOP, Dist);
76            %increment generation counter
77            gen=gen+1;
78          end
79          toc;
80          minimum
81  end
```

3.3 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
```

3.4 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, :);
      NewChrom(row+1,:) = MatrixChrom(2, :);
   else
      NewChrom(row,:) = OldChrom(row,:);
      NewChrom(row+1,:) = OldChrom(row+1,:);
   end
    end

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

% End of function
```

3.5 order_low_level.m

```
% low level function for calculating an offspring
% given 2 parent in the Parents - agrument
% Parents is a matrix with 2 rows, each row
% represent the genocode of the parent
%
% Returns a matrix containing the offspring


function Offspring=order_low_level(Parents)

    cols = size(Parents,2);

    Offspring=zeros(2,cols);

    start_index = rand_int(1, 1, [1, cols - 1]);
    end_index = rand_int(1, 1, [start_index + 1, cols]);

    Offspring(1, start_index:end_index) = Parents(2, start_index:
       end_index);
    Offspring(2, start_index:end_index) = Parents(1, start_index:
       end_index);


    for off=1:2
        Buff = Parents(off,:);
        Buff = [Buff(end_index+1:end), Buff(1:end_index)];

        members = ismember(Buff, Offspring(off, :));
        Buff(members == 1) = 0;

        ii = 1;
        X = find(Buff);
        for jj=1:start_index - 1
            if Buff(X(ii)) ~= 0
                Offspring(off, jj) = Buff(X(ii));
                Buff(X(ii)) = 0;
                ii = mod(ii, cols) + 1;
            end
        end

        ii = 1;
        X = find(Buff);
        for jj=end_index + 1:cols
```

```
            if Buff(X(ii)) ~= 0
                Offspring(off, jj) = Buff(X(ii));
                Buff(X(ii)) = 0;
                ii = mod(ii, cols) + 1;
            end
        end
        %Offspring(off, end_index+1:end) = Buff(start_index:end);
        %Offspring(off, 1:start_index - 1) = Buff(1:start_index -
            1);
    end
% end function
```

3.6 tspgui.m

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

3.7 tspfun.m

```
%
% ObjVal = tspfun(Phen, Dist)
% Implementation of the TSP fitness function
% Phen contains the phenocode of the matrix coded in path
% representation
% 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
```

3.8 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
```

```
29        NewChrom(r,:) = feval(MUT_F, OldChrom(r,:));
30    end
31 end
32
33 % End of function
```