

KU LEUVEN

GENETIC ALGORITHMS

---

**TSP**

---

*Author:*

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

January 7, 2017

# Contents

<b>1</b>	<b>Experiments</b>	<b>2</b>
<b>2</b>	<b>Implementation</b>	<b>2</b>
2.1	Representation . . . . .	2
2.2	Crossover . . . . .	2
2.3	Mutation . . . . .	3
2.4	Fitness Function . . . . .	3
<b>3</b>	<b>Appendix</b>	<b>5</b>
3.1	tsp_ImprovePopulation.m . . . . .	5
3.2	run_ga.m . . . . .	6
3.3	insertion.m . . . . .	8
3.4	order_crossover.m . . . . .	8
3.5	order_low_level.m . . . . .	10
3.6	tspgui.m . . . . .	11
3.7	tspfun.m . . . . .	11
3.8	mutateTSP.m . . . . .	12

# 1 Experiments

## 2 Implementation

### 2.1 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`.

### 2.2 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:

---

<sup>1</sup>This explanation can be found in the slides about Traveling Salesman Problem

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

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

$$\begin{array}{c} (1\ 2\ \text{---}\ 3\ 4\ 5\ \text{---}\ 6\ 7\ 8) \\ (2\ 4\ \text{---}\ 6\ 8\ 7\ \text{---}\ 5\ 3\ 1) \end{array}$$

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:

$$\begin{array}{c} (8\ 7\ \text{---}\ 3\ 4\ 5\ \text{---}\ 1\ 2\ 6) \\ (4\ 5\ \text{---}\ 6\ 8\ 7\ \text{---}\ 1\ 2\ 3) \end{array}$$

## 2.3 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 deecided 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:

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

## 2.4 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.

## 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

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

## 3.2 run\_ga.m

```
1 function run_ga(x, y, NIND, MAXGEN, NVAR, ELITIST, STOP_PERCENTAGE
  , PR_CROSS, PR_MUT, CROSSOVER, LOCALLOOP, ah1, ah2, ah3)
2 % usage: run_ga(x, y,
3 %           NIND, MAXGEN, NVAR,
4 %           ELITIST, STOP_PERCENTAGE,
5 %           PR_CROSS, PR_MUT, CROSSOVER,
6 %           ah1, ah2, ah3)
7 %
8 %
9 % x, y: coordinates of the cities
10 % NIND: number of individuals
11 % MAXGEN: maximal number of generations
12 % ELITIST: percentage of elite population
13 % STOP_PERCENTAGE: percentage of equal fitness (stop criterium)
14 % PR_CROSS: probability for crossover
15 % PR_MUT: probability for mutation
16 % CROSSOVER: the crossover operator
17 % calculate distance matrix between each pair of cities
18 % ah1, ah2, ah3: axes handles to visualise tsp
19 {NIND MAXGEN NVAR ELITIST STOP_PERCENTAGE PR_CROSS PR_MUT
  CROSSOVER LOCALLOOP};
20
21 tic ;
22 GGAP = 1 - ELITIST;
23 mean_fits=zeros(1,MAXGEN+1);
24 worst=zeros(1,MAXGEN+1);
25 Dist=zeros(NVAR,NVAR);
26 for i=1:size(x,1)
27     for j=1:size(y,1)
28         Dist(i,j)=sqrt((x(i)-x(j))^2+(y(i)-y(j))^2);
29     end
30 end
31 % initialize population
32 Chrom=zeros(NIND,NVAR);
33 for row=1:NIND
34     %Chrom(row,:)=path2adj(randperm(NVAR));
35     Chrom(row,:)=randperm(NVAR);
36 end
37 gen=0;
38 % number of individuals of equal fitness needed to stop
39 stopN=ceil(STOP_PERCENTAGE*NIND);
40 % evaluate initial population
41 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,
57         gen, best, mean_fits, worst, ah2, ObjV, NIND, ah3);
58         visualizeTSP(x,y,Chrom(t,:), minimum, ah1, gen, best,
59         mean_fits, worst, ah2, ObjV, NIND, ah3);
60
61         if (sObjV(stopN)-sObjV(1) <= 1e-15)
62             break;
63         end
64         %assign fitness values to entire population
65         FitnV=ranking(ObjV);
66         %select individuals for breeding
67         SelCh=select('sus', Chrom, FitnV, GGAP);
68         %recombine individuals (crossover)
69         SelCh = recomb(CROSSOVER,SelCh,PR_CROSS);
70         %SelCh=mutateTSP('inversion',SelCh,PR_MUT);
71         SelCh=mutateTSP('insertion',SelCh,PR_MUT); % <— line
72         changed, now insertion mutation is used
73         %evaluate offspring, call objective function
74         ObjVSel = tspfun(SelCh,Dist);
75         %reinsert offspring into population
76         [Chrom, ObjV]=reins(Chrom,SelCh,1,1,ObjV,ObjVSel);
77
78         Chrom = tsp_ImprovePopulation(NIND, NVAR, Chrom,
79         LOCALLOOP,Dist);
80         %increment generation counter
81         gen=gen+1;
82     end
83     toc;
84     minimum
85 end

```



### 3.3 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 end
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
```

### 3.4 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).
```

```

7 %      XOVR      – Probability of recombination occurring between
      pairs
8 %
      of individuals.
9 %
10 % Output parameter:
11 %      NewChrom – Matrix containing the chromosomes of the
      population
12 %
      after mating, ready to be mutated and/or
      evaluated,
13 %
      in the same format as OldChrom.
14 %
15
16 function NewChrom = order_crossover (OldChrom, XOVR)
17
18 if nargin < 2, XOVR = NaN; end
19 [rows,~]=size (OldChrom) ;
20
21     maxrows=rows;
22     if rem(rows,2)~=0
23         maxrows=maxrows-1;
24     end
25
26     for row=1:2:maxrows
27
28         % crossover of the two chromosomes
29         % results in 2 offsprings
30         if rand<XOVR      % recombine with a given probability
31             MatrixChrom = order_low_level ([OldChrom(row,:) ;OldChrom(
                row+1,:) ] ) ;
32             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
39
40     if rem(rows,2) ~= 0
41         NewChrom(rows,:)=OldChrom(rows,:) ;
42     end
43
44 % End of function

```

### 3.5 order\_low\_level.m

```
1 % low level function for calculating an offspring
2 % given 2 parent in the Parents - argument
3 % Parents is a matrix with 2 rows, each row
4 % represent the genotype 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);
20     Offspring(2, start_index:end_index) = Parents(1, start_index:
21         end_index);
22
23     for off=1:2
24         Buff = Parents(off,:);
25         Buff = [Buff(end_index+1:end), Buff(1:end_index)];
26
27         members = ismember(Buff, Offspring(off, :));
28         Buff(members == 1) = 0;
29
30         ii = 1;
31         X = find(Buff);
32         for jj=1:start_index - 1
33             if Buff(X(ii)) ~= 0
34                 Offspring(off, jj) = Buff(X(ii));
35                 Buff(X(ii)) = 0;
36                 ii = mod(ii, cols) + 1;
37             end
38         end
39
40         ii = 1;
41         X = find(Buff);
42         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
% end function

```

### 3.6 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);

```

### 3.7 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
    tour
8 % (=each row of Phen)
9 %
10
11 function ObjVal = tspfun(Phen, Dist)
12     % the objective function works with adjacency representation.
    In this
13     % version, path representation is used, so the fitness
    function should
14     % be adapted. Now, the phenotype is converted to adjacency
    % representation first, and then, the Objective Value is
15     % computed as it
16     % was computed in the original version.
17     adj = zeros(size(Phen));

```

```

18     for row=1:size(Phen)
19         adj(row,:) = path2adj(Phen(row,:));
20     end
21
22     ObjVal = Dist(adj(:,1), 1);
23     for t=2:size(adj,2)
24         ObjVal=ObjVal + Dist(adj(:,t), t);
25     end
26
27 % End of function

```

### 3.8 mutateTSP.m

```

1 % MUTATE_TSP.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(MUT_F, OldChrom, MutOpt)
8 %
9 % Input parameter:
10 %   MUT_F      - 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 % Output parameter:
17 %   NewChrom    - Matrix containing the chromosomes of the
18 %                population
19 %                after mutation in the same format as OldChrom.
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
```