Genetics algorithm and evolutionary computing

Robbe Keppens,Afraz Salim

January 6, 2019

Abstract

This report illustrates different components of a genetic algorithm for the Traveling Salesman problem. We investigate the various effects of different parameters on the efficiency and effectiveness of the algorithm. This is done both for the original adjacency representation based algorithm and an alternative path-representation based algorithm. We also check the benefits and drawbacks of a stopping criterion, and the efficiency gains of a local optimization heuristic. Finally, we also provide a performance comparison of these algorithm for different benchmark problems.

Experiments on Genetic algorithm by varying it's parameters

This section shows the impact of changing the parameters of the genetic algorithm. It is a known fact that diversity is required to attain a better solution. Two variation operators, mutation and recombination, are of great interest in this matter.

Mutation

The minimum distance that can be found for 127 cities, with 1000 generations, is 19.0206. It is possible that this is not the minimum distance but this value is sufficient to perform some tests.

The following tests are performed in order to find the **Mean-Best-Fitness**. Experiments are performed with different mutation rate and each experiment is performed 5 times. Data used for the tests:

cities	individuals	Generations	pr.mutation	p.crossover	elite
127	50	300	between[0-100]	95	5

Result of these experiments can be seen in 1.

Conclusion about mutation rate

One can conclude that setting the mutation rate high creates too many disruptions, preventing the algorithm from converging to an optimal solution as it keeps generating bad mutations. However, the optimal mutation rate depends upon the survivor selection method. Elitism, for example, seems to be a great mutation strategy as it preserves one or more of the best members in each generation, ensuring that progress is not lost.

On the other hand, setting a low mutation rate means that one must take additional measures to ensure population diversity. If that is not done, then low mutation rate can result in premature convergence, meaning that the genetic algorithm gets stuck on a local optimum from which it can not escape.

Crossover Rate

In this section the effect of crossover rate is discussed by performing experiments. The following data is used to perform experiments:

cities	individuals	Generations	pr.mutation	p.crossover	elite
127	50	500	30	between [5-100]	5

Each experiment is performed 5 times and the median value is used as final value.The result of this experiment can be seen in 2. It seems that smaller values for the crossover rate perform better but then the algorithm requires more iterations to improve because during each generation a very limited set of edges are exchanged. However, as the generations increase, the algorithm starts converging to the optimal solution.

Implementation of a stopping criteria

Choosing the best stopping criteria for a genetic algorithm is one of the hardest tasks.A simple way to implement a stopping criteria is by imposing a constraint on the number of generations, but there are few problems with this approach.

The technique does not consider the state of the population, such as it's diversity or the quality of this solution. It relies entirely on a user-chosen number, which is often based purely on guesswork. If the number chosen is too large, the algorithm performs unnecessary and unproductive work. If the number chosen is too small, then the algorithm may be ended before it can converge to a proper solution.

Avoiding useless iterations

An alternative technique can be used to avoid useless iterations. It is not efficient to check the diversity after each generation but the information from last n generations can be used to decide whether the algorithm is improving the solution or not. More specifically, if there has been no improvement during the last n iterations then the algorithm should stop.A good value for n can be found experimentally, which will be discussed later.

Two experiments were performed to check the effectiveness of this technique. The first experiment was conducted with the following parameters:

cities	individuals	Generations	pr.mutation	p.crossover	elite
127	50	1000	5	95	5

The second experiment used the same parameters as the first one but imposed a constraint. If no improvement was detected in the last 100 iterations then algorithm was stopped, even if it hadn't yet reached it's maximum number of generations.The result of these experiments are shown in 3 and 4 respectively.

Conclusion

With the parameters listed above,best length of the route with 1000 iterations is 22.6724 while this result ($22.6724 \approx 22.954$) was obtained using only 231 iterations. The last 100 generations did not see any performance so the algorithm stopped. Not immediately stopping the algorithm once it shows no improvement gives it a significant amount of chances to show further improvement, without doing an excessive amount of unnecessary work.

There are also certain drawbacks of this approach. Due to the random nature of the algorithm, it can be the case that the algorithm keeps repeatedly exploiting some regions and before it explores new regions where better results may be obtained. If the N chances have been used for exploitation instead of exploration then the algorithm will declare a sub optimal solution as final solution. As such, determining the value of N is a balancing act between avoiding unnecessary work and risking potential improvements.

Alternative representation and appropriate operators

Although the representation of the members of the population has no direct effect on the genetic algorithm, the representation does determine the ease and efficiency by which various operators can be implemented. As such, the choice of representation method restricts the potential options for mutation and crossover operators.

The original algorithm utilized adjacency representation. A variant has been included which instead works with path representation. In both path and adjacency representation, the cities are represented as a list of n cities. In adjacency representation, the position in the list represents the city, whereas the value in that position represents the next city to be visited. In path representation, the place in the list represents the order in which cities are visited, and the value stored represents which city is visited.

Path representation is a more natural way to represent the information, as it allows people to visualize the path at a glance.

Mutation Operators

There are 3 mutation operators. Two mutation operators were already implemented ;"reciprocal exchange" and "inversion". These have been supplemented by a third mutation operation operator; the "cut" mutation. Reciprocal exchange switches two cities, while inversion inverts a random subtour. Cut mutation removes one subtour from the path, and injects it at a different location.

Crossover Operators

Only 1 crossover operator has been implemented, which is Order Based Crossover. In order based crossover, a number of cities are selected in one parent. The order in which these cities are ranked is then forced upon the other parent.

Parameter Optimization

There are many parameters in a genetic algorithm which can be optimized. For this parameter optimization, we run the algorithm in the same configuration, except for 1 varying variable. We pick the optimal result, and then move on to the next parameter being optimized.

The parameters being optimized are the mutation rate, crossover rate, the size of the population and number of generations..

Mutation Method

Three mutation methods were programmed (cut, inversion and reciprocal exchange). These methods were compared on rondtrip127.tsp, with no loop detection, 200 individuals, 500 generations, 5% mutation rate, 95% crossover chance, a 5 % elite.

Method	time	distance
Cut	96	13.586
Inversion	104	14.9907
Reciprocal exchange	99	13.388
Cut+Inversion	97	14.5452
Cut+Reciprocal exchange	97	14.4251
Inversion+Reciprocal exchange	101	14.0649

The execution times are almost identical,indicating that the various operators have no significant effect on performance. The distances acquired do vary somewhat between methods. The best method appears to be Reciprocal exchange, and is thus the method that will be used in the following experiments.

Utilization of multiple mutation methods simultaneously appears to be counterproductive.

Mutation rate

This experiment uses the same parameters as the previous experiment, but varies the mutation chance between 5 and 30%. As shown in the table, increasing the mutation rate at first increases the results, but after that rapidly decreases the performance of the algorithm because the population deviates too rapidly from promising solutions.

pr.mutation	time	distance
5	99	13.338
10	99	12.8705
15	99	13.7843
20	99	13.626
30	100	17.3072

Crossover rate

This experiment utilized the same parameters as the previous test and a 10% mutation rate. The results appear to be optimal at the current 95% mutation rate. Both below and above the performance declines.

pr.crossover	time	distance
80	100	13.8092
90	98	13.8272
95	99	12.8705
100	99	15.0811

Population

In this experiment, the effect of population size was tested. The same map and parameters as the previous tests were utilized. Population sizes of 100, 200, 400 and 600 were tested.

As can be expected, increasing the population size increases the run time of the program. It also provides only small improvements in performance. As such, a population size of 200 seems optimal.

Population	time	distance
100	101	13.952
200	99	12.8705
400	112	12.9681
600	122	12.5801

Generations

In this experiment, the effect of generations was tested. The same map and parameters as the previous tests were utilized. Generation sizes of 100, 500 and 1000 were tested.

Population	time	distance
100	22	21.2023
500	112	12.8705
1000	214	10.6775

As can be expected, performance improves rapidly at first before slowing down considerably. 500 generations provides a decent equilibrium between speed of computation and accuracy of result.

Conclusions

The behaviour of the modified version of the algorithm differs significantly from the original version. In the original version, the average and worst members of population where significantly less fit than best members of the population. All 3 engaged in a relatively stable decline.

The modified algorithm, as illustrated in 7, demonstrates a significantly more erratic pattern. Average fitness holds closely to the best members of the population, except for regularly occurring sporadic jumps .

The origin of this behaviour is the chosen crossover operator. While initially not a problem, it caused significant degradation of effectiveness when the running the benchmark problems. Because of that, it was decided to replace the crossover operator with cycle-crossover.m.

Performance of algorithm based of benchmarks

In this section we analyze the performance of the algorithm, based on experiments performed on the problems for which we know the optimal path length. The Operators used in this section are listed below:

Crossover op	mutation operator	Parent Selection	Survivor Selection
cycle-cross-over op	cut-mutation	Tournament Selection	Elitism

The first problem involves solving the tsp problem for 131 cities. The following results are obtained with the corresponding parameters.

Individuals	time	Generation	mutation rate	cross-over rate	optimal length	obtained length
400	207 sec	1000	20	55	564	751

A screenshot of this result can also be seen in 6.

Benchmark problem with number of cities 380

Initial distance for this problem is 24003. The parameters and result are listed in following table:
The original distance is 23631:

Individuals	time	Generation	mutation rate	cross-over rate	optimal length	obtained length
600	832 sec	1500	20	55	1621	4262

Benchmark problem with number of cities 662

. This problem seems to show the worst solution from all the benchmarks problem. The difference between the optimal solution and obtained solution is greatest in this problem. From visual analysis it appears that the population converge to one solution very quickly The lost diversity prevents the algorithm from effectively improving the results.

Individuals	time	Generation	mutation rate	cross-over rate	optimal length	obtained length
600	1308 sec	1500	20	55	2513	13633

Benchmark problem with number of cities 711

The optimal distance for this problem is 3115 with number of cities 711. Search space is huge and in order to avoid the chances of losing diversity , we have to keep more individuals in pool. Experiments were performed with following parameter and last column indicates the obtained result within those number of generations.
Initial distance of this problem is : 61034.

Individuals	time	Generation	mutation rate	cross-over rate	optimal length	obtained length
600	1004 sec	2500	20	55	3115	9564

It seems from the experiment that if we allow more generations then the algorithm keeps improving the population.The chosen cross-over rate plays a crucial role at certain points. Choosing a low cross-over rate moves the population quickly towards the optimal value but if the cross-over rate is not chosen according to the problem size then all individuals converge to one solution before obtaining the optimal value. This happens because limited number of individuals represent limited diversity.

Belgium tour problem with number of cities 41

. Following parameters were used to perform the experiments and again the obtained length indicates the best path found.

Individuals	time	Generation	mutation rate	cross-over rate	optimal length	obtained length
600	20 sec	400	15	25	not-known	720

Test of a local optimization heuristic

The local optimization heuristic improves our genetic algorithm by evaluating each member of the population and potentially making a small change to improve their fitness. By doing so, the heuristic restricts the search space, focusing the genetic algorithm on a smaller but fitter subsection of the search space.

This was tested using both representations, rondrit127.tsp with 200 individuals, 500 generations, 15% mutation rate, 95% crossover and a 5% elite. Path representation used Reciprocal exchange mutator and either the order based crossover or the cycle based crossover. Adjacency representation used inversion and alternate edges crossover.

Loop	time	distance
No - Path (OX2)	101	12.7834
Yes - Path (OX2)	106	10.896
No - Path (cycle)	133	19.6734
Yes - Path (cycle)	128	15.5145
No - Adjacency	133	18.928
Yes - Adjacency	127	10.514

The result is a considerable improvement in all cases, and no meaningful cost in run time. This improvement is especially pronounced with Adjacency representation, where the local loop optimization manages to surpass the final non-locally optimized result within 6 generations. The path representation using a cycle crossover also saw similar rapid improvement. For path representation (OX2), the effect is less pronounced but still considerable.

Implementation of Parent Selection methods

In this section two different parent selection strategies, namely tournament selection and ranking selection, are discussed and their performance is evaluated.

Tournament selection

Tournament selection works by picking K individuals and compare them against each other to select the best one from those K individuals and denoting it ith candidate. This process is repeated until μ individuals are found.

Code for tournament selection is given in 1.

Ranking Selection

Ranking selection is based on sorting the population according to their fitness and then assigning new selection probabilities according to their rank. These probabilities are then used as fitness to select the parents for selection. To compute the selection probabilities , following formula is used:

$P(i) = \frac{(2-S)}{\mu} + \frac{2i(S-1)}{\mu(\mu-1)}$

Code for Ranking Selection is shown in 2.

Comparison between algorithms

We further evaluate the performance of tournament selection. The following parameters are used to compare the performance of three selection method.

pr.mutation	cities	individuals	Generations
5	127	100	500

Each experiment is performed 5 times with same parameters and the mean value is used for final result.However , mean value does not exist always but a value is picked from a certain interval which contains more nearby values. A comparison between tournament selection,ranking selection, stochastic universal sampling can be seen in 5. That graph is based on the following data:

Cities	SUS	TS	RS
127	20.017	20.4155	19.5455
70	19.3816	18.18	19.0655
50	12.204	12.09	11.1245
25	4.6242	4.4284	4.6465
16	3.4328	3.35	3.35

Tournament selection preserves a high selection pressure and this can lead to the premature convergence. Ranking selection changes the selection probabilities according to their rank and this gives also some less fitter individuals a chance to get selected but this algorithm converges slowly and the chances of premature convergence are also small. A second comparison needs to be made about the computation of these probabilities. Ranking selection is computationally expensive.

Conclusion

Every operator along with the chosen value has significant impact on the performance of the algorithm. One conclusion obtained from benchmarks results and parameter analysis suggests that rate of the cross-over operator and mutation operator should be chosen according to the problem. Choosing smaller values pushes the individuals toward the minimum value quiet quickly but it can result in premature convergence if the rate of these operators is chosen small w.r.t size. On the other hand, chosing very high values for these operators results in many different individuals but the algorithm requires more generations on average.

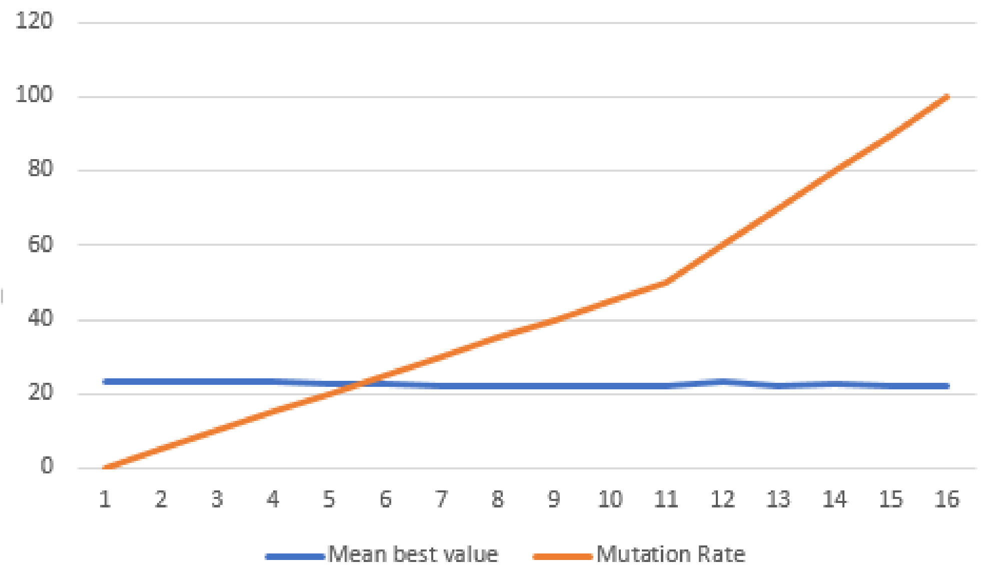

Figure 1: A comparison of mutation rate and mean best fitness on 300 generations.

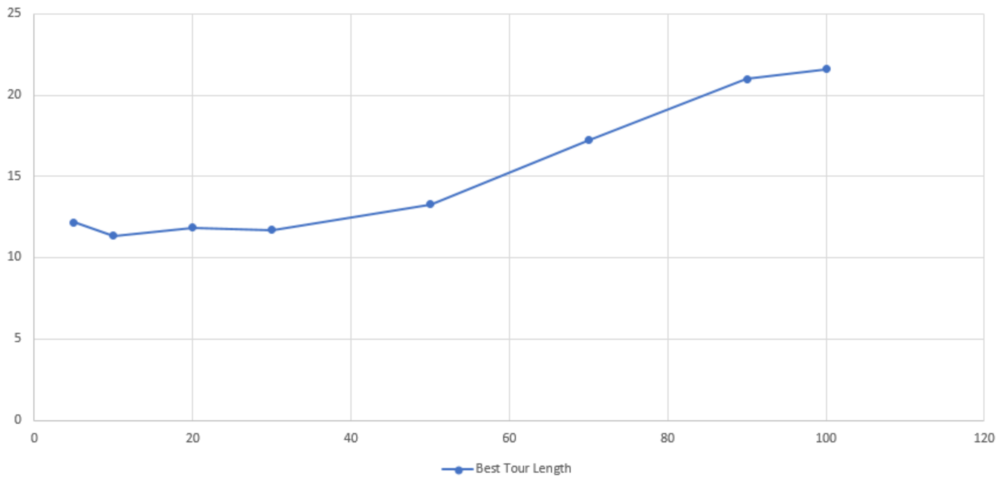

Figure 2: A comparison of cross over rate and mean best fitness on 300 generations.

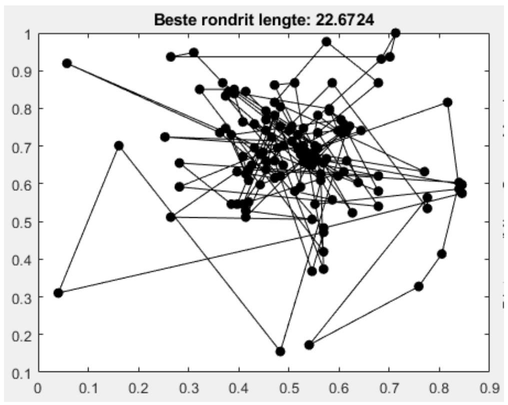

Figure 3: Best route after 1000 iterations.

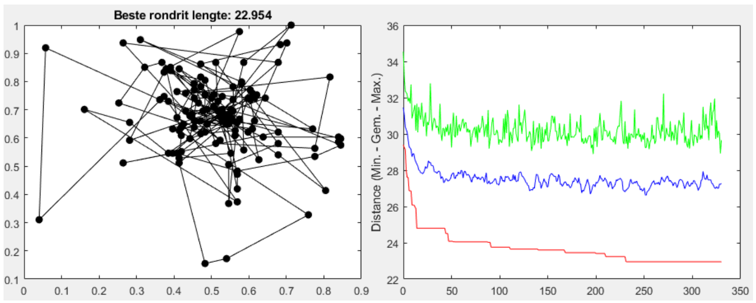

Figure 4: Best route after 331 iterations.

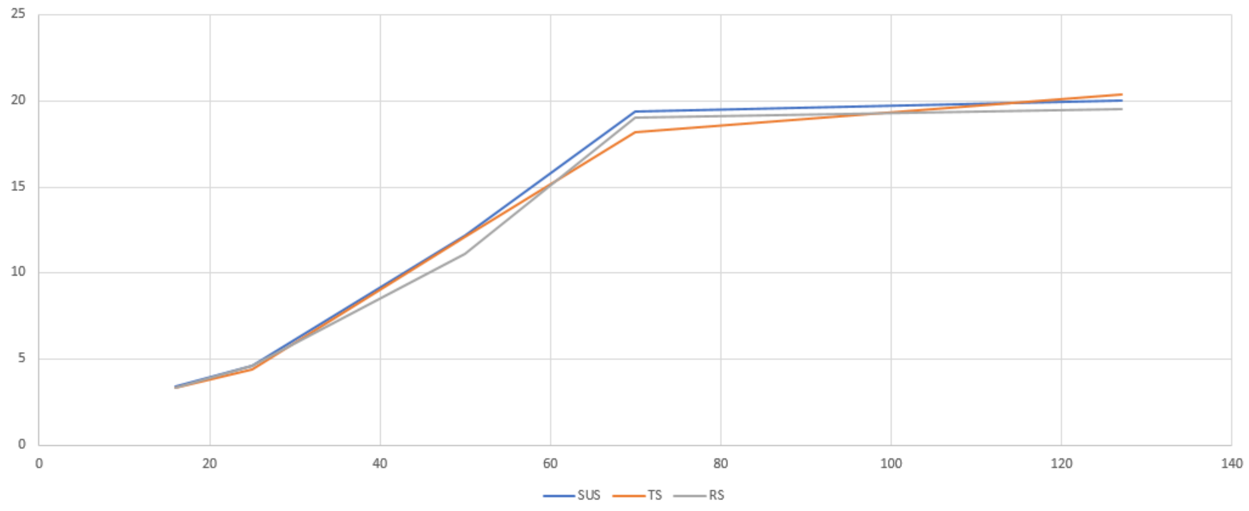

Figure 5: A comparison between different selection algorithms. Cities are on x-axis while y-axis indicate the distance.

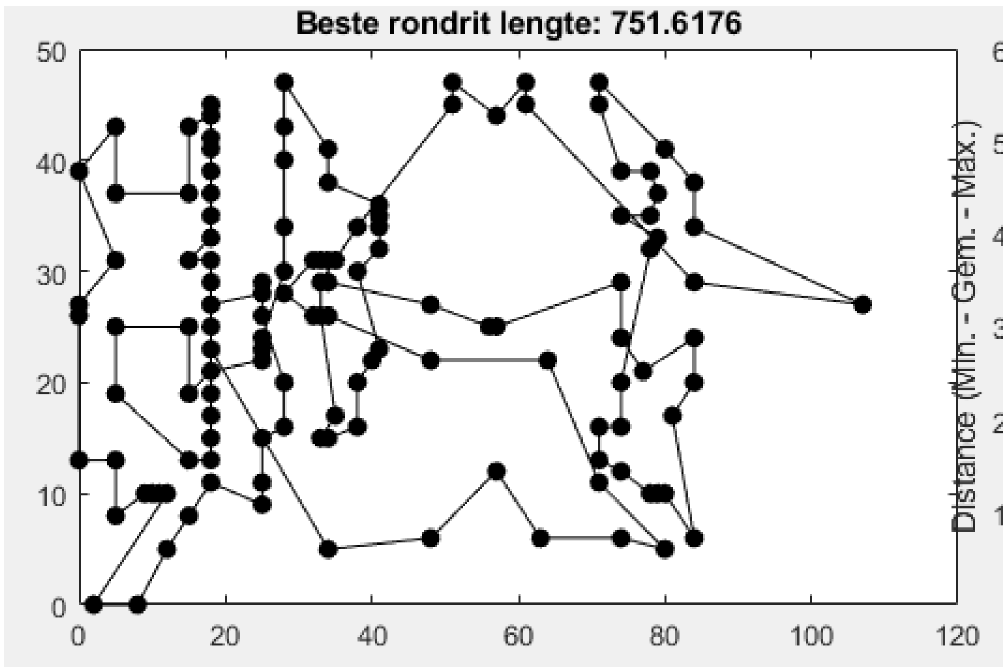

Figure 6: Result obtained for first benchmark problem.

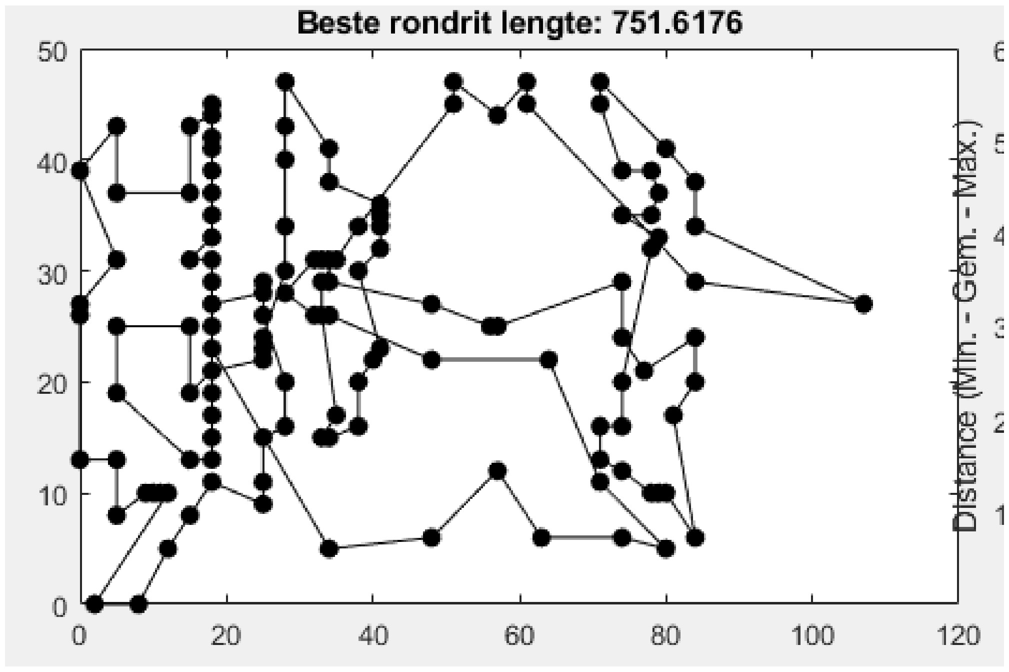

Figure 7: This image shows the behaviour of the path-based algorithm.

Code for stopping criteria

```
classdef improvement
    properties
      previousCounter = 0;
      previousDistance = 0;
      result = false;
    end
    methods
     function r = improvementGen(value,distvalue,maxChances)
         resultValue = abs(distvalue-value.previousDistance);
           if (resultValue > 0)
               value.previousDistance = distvalue;
               value.result = true;
               value.previousCounter = 0;
               r = value;
           else
               value.previousCounter = value.previousCounter + 1;
               if(value.previousCounter >= maxChances)
                   value.result = false;
                   r = value;
               else
                   value.result = true;
                   r = value;
               end
           end

     end
    end
   end
```

Listing 1: Tournament Selection

```
%numberOfCandidates: How many candidates take part in competition.
function SelCh = tournamentSelect(Chrom, FitnV, GGAP,SUBPOP,numberOfCandidates);

% Check parameter consistency
   if nargin < 3, error('Not enough input parameter'); end

   % Identify the population size (Nind)
   [NindCh,Nvar] = 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

   if nargin < 5, SUBPOP = 1; end
   if nargin > 4,
      if isempty(SUBPOP), SUBPOP = 1;
      elseif isnan(SUBPOP), SUBPOP = 1;
      elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
   end

   if (NindCh/SUBPOP) ~= fix(NindCh/SUBPOP), error('Chrom and SUBPOP disagree'); end
   Nind = NindCh/SUBPOP;  % Compute number of individuals per subpopulation
   if nargin < 4, GGAP = 1; end
   if nargin > 3,
      if isempty(GGAP), GGAP = 1;
      elseif isnan(GGAP), GGAP = 1;
      elseif length(GGAP) ~= 1, error('GGAP must be a scalar');
      elseif (GGAP < 0), error('GGAP must be a scalar bigger than 0'); end
   end

% Compute number of new individuals (to select)
   NSel=max(floor(Nind*GGAP+.5),2);

% Select individuals from population
  SelCh = [];
  currentIndex = 1;
  [ m,n] = size(FitnV);
   for irun = 1:NSel,
       %Generate random number between 0-Nind
       r = randi(m,1,numberOfCandidates);
       currentIndex = r(1);
       for j = 1:numberOfCandidates-1,
         if FitnV(r(j)) > FitnV(r(j+1))
             currentIndex = r(j);
         else
             currentIndex = r(j+1);
         end
       end
       SelCh=[SelCh;Chrom(currentIndex,:)];
   end
```

Listing 2: Ranking Selection

```
function SelCh = rankingSelection(SEL_F, Chrom, FitnV, GGAP,S);

% Check parameter consistency
   if nargin < 3, error('Not enough input parameter'); end

   % Identify the population size (Nind)
   [NindCh,Nvar] = 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

   if nargin < 5, SUBPOP = 1; end
   if nargin > 4,
      if isempty(SUBPOP), SUBPOP = 1;
      elseif isnan(SUBPOP), SUBPOP = 1;
      elseif length(SUBPOP) ~= 1, error('SUBPOP must be a scalar'); end
   end

   if (NindCh/SUBPOP) ~= fix(NindCh/SUBPOP), error('Chrom and SUBPOP disagree'); end
   Nind = NindCh/SUBPOP;  % Compute number of individuals per subpopulation

   if nargin < 4, GGAP = 1; end
   if nargin > 3,
      if isempty(GGAP), GGAP = 1;
      elseif isnan(GGAP), GGAP = 1;
      elseif length(GGAP) ~= 1, error('GGAP must be a scalar');
      elseif (GGAP < 0), error('GGAP must be a scalar bigger than 0'); end
   end

% Compute number of new individuals (to select)
   NSel=max(floor(Nind*GGAP+.5),2);

% Select individuals from population
   Pop = [];
   Ind = [];
   for k = 1:NindCh,
       Index = 1;
       %Sort the entire Population according to the fitness level.
       for j = 1: size(FitnV) ,
           if FitnV(j) < FitnV(Index)
               Index = j;
           end
       end
       Ind  = [Ind,Index];
       Pop = [Pop; Chrom(Index,:)];
       FitnV(Index) = [];
       Chrom(Index,:) = [];
   end
   SelProb = zeros(NindCh,1);
   S = 1.5
```

Listing 3: Cycle Crossover

```
function Offspring=cycle_crossOver(Parents);
        cols=size(Parents,2);
        %Offspring=Parents(2,:);
    ParentFirst = Parents(1,:);
    ParentSecond = Parents(2,:);
    randomNumbers = rand(1,cols);
    Offspring = [];
    Contain = zeros(1,cols);
    for i= 1:cols
        if randomNumbers(i) >= 0.5
            for k =1:cols
               result =  Contain(1,ParentFirst(1,k));
               if result == 0
                  Offspring = [Offspring,ParentFirst(1,k)];
                  Contain(1,ParentFirst(1,k))= 1;
                  break;
               end
            end
        else if randomNumbers(i) < 0.5
              for p =1:cols
               result =  Contain(1,ParentSecond(1,p));
                 if result == 0
                     Offspring = [Offspring,ParentSecond(1,p)];
                     Contain(1,ParentSecond(1,p))= 1;
                     break;
                end
            end
            end
        end
    end
      if length(Offspring) ~= length(unique(Offspring))
           warning('something went wrong');
      end
end
```

Listing 4: Order based Crossover

```
function Offspring=cross_order_based(Parents);

        cols=size(Parents,2);
        Offspring=Parents(2,:);
    positions = sort(randperm(cols, floor(cols/3))); % Create an array with unique numbers to

    cities=Parents(1, positions);

    mask = ismember(Offspring(1,:), cities) ;

    k = 1;
    for j=1:cols
        if mask(j)
            Offspring(1,j)= cities(k);
            k = k + 1;
        end
    end
end
```

Listing 5: Cut Mutation

```
function NewChrom = cut(OldChrom,Representation);

NewChrom=OldChrom;

if Representation==1
        NewChrom=adj2path(NewChrom);
end

length = 3;
start = randi(size(NewChrom,2)-length);
cut = NewChrom(start : start+length);
NewChrom(start : start+length) = [];
insert = randi(size(NewChrom,2));
if insert==1
    NewChrom = [cut NewChrom(insert: size(NewChrom,2))];
else
    NewChrom = [NewChrom(1:insert-1) cut NewChrom(insert: size(NewChrom,2))];
end

if Representation==1
        NewChrom=path2adj(NewChrom);
end


% End of function
```

Listing 6: path fitness function

```
function ObjVal = tspfun_path(Phen, Dist)
    ObjVal=zeros(size(Phen,1),1);
    for k = 1 : size(Phen,1)
        ObjVal(k,1)=Dist(Phen(k,1),Phen(k,end)); % Add distance from last element back to firs
        for t=1:(size(Phen,2)-1)
            ObjVal(k,1)=ObjVal(k,1)+Dist(Phen(k,t),Phen(k,t+1));
% Calculate distance for every part of the path
        end
    end
```

Listing 7: Modified algorithm for path representation

```
function run_ga_path(x, y, NIND, MAXGEN, NVAR, ELITIST, STOP_PERCENTAGE, PR_CROSS, PR_MUT, CRO
% 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 % Timer function to evaluate performance

        maxChances = 1000;
        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);  % CHROM now contains n paths of NVAR length
        end
        gen=0;
        % number of individuals of equal fitness needed to stop
        stopN=ceil(STOP_PERCENTAGE*NIND);
        % evaluate initial population
        ObjV = tspfun_path(Chrom,Dist);
        best=zeros(1,MAXGEN);
        %Create a new StopingCriteria's class
        impObject = improvement;
        %imp.previousCounter = 0;
        % generational loop
        while (gen<MAXGEN)
```