KU Leuven

Genetic Algorithms

TSP

Author:

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

January 6, 2017

Contents

1 Experiments

no loop detection, 50 individuals, 100 generations, 5 pr mutation, 95 pr crossover, 5% elite.
16 cities 5.328203 3.5417 8.262045 3.7098 8.712387 3.6799 8.569090 3.5928
Same with 18 cities 14.058956 3.526 9.053208 3.4481 8.518912 3.1463 8.703639 3.4115
23 cities 8.469464 4.4202 8.421920 4.9056 8.751834 4.9177 8.352938 4.9312
25 cities 9.033588 6.4734 8.593667 6.0601 8.544754 6.4646 9.093625 6.0974
48 cities 9.284919 11.3248 8.837234 11.1833 8.765404 11.3676 9.322479 11.4889
50 cities 9.372256 15.883 9.060394 15.8207 8.916035 16.8945 9.168466 16.9276
51 cities 8.821235 15.6717 8.918025 15.5546 9.583848 14.8687 9.635809 15.8326
67 cities 9.603053 15.6171 9.084740 16.1501 9.176364 15.6805 9.108434 16.27
70 cities 9.888628 25.9827 10.043178 25.2098 9.517789 24.5681 9.443164 24.6575
100 cities 10.273545 39.1235 9.884619 39.5773 10.742947 37.024 9.888532 39.4329
127 cities 10.772273 23.7809 10.371431 24.283 10.233814 24.0806 10.217393 23.8229s

2 Apendix

- Path representation - Order Crossover (http://www.dca.fee.unicamp.br/ẽomide/courses/EA072/artigos/
) - Insertion mutation (http://mnemstudio.org/geneticalgorithmsmutation.htm)

2.1 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;
```

2.2 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);
        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;
```

```
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
```

2.3 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
```

```
    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
```

2.4 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
```

2.5 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)
```

```
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
```

2.6 tspgui.m

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

2.7 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
```

2.8 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
```