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

2 Implementation

2.1 Representation

The origial code employed adjacency representation. EXPLAIN ADJACENCY REPRESENTATION AND EXAMPLE

In our implementation, we have deceided to use path representation. EXPLAIN PATH REPRESENTATION AND EXAMPLE. GIVE REASONS

2.2 Mutation

- Insertion mutation (http://mnemstudio.org/geneticalgorithmsmutation.htm) EXPLAIN WHAT MUTATION IS

The mutation operator selected for this problem is insertion mutation. EXPLAIN HERE WHY AND WHAT IT IS

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

2.3 Crossover

- Order Crossover (http://www.dca.fee.unicamp.br/g̃omide/courses/EA072/artigos/Genetic_Algorithm_T
)

EXPLAIN WHAT CROSSOVER IS

EXPLAIN ORDER CROSSOVER

EXAMPLE

2.4 Fitness Function

The fitness functio has been changed

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);
        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
              SelCh=select('sus', Chrom, FitnV, GGAP);
              %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
```

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'; % default crossover operator}
    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
     NewChrom(r,:) = feval(MUT_F, OldChrom(r,:));
  end
end

% End of function
```