Home assignment 3 (ETS061) - *Simulation*

Niclas Lövdahl <dic13nlo@student.lu.se>

Task 1

Results from simulation. (loadatt48.m)

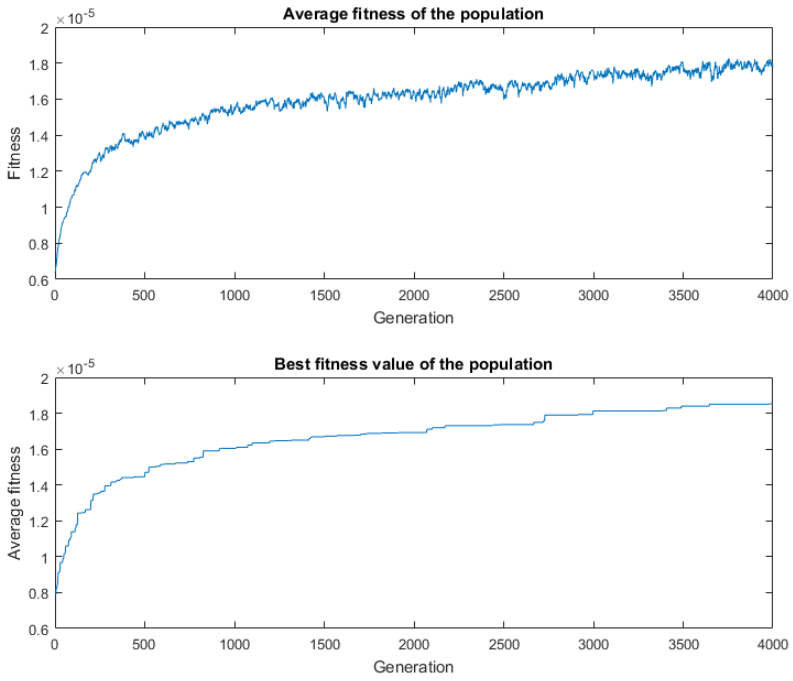

Figure 1: Simulation of 4000 generation with data from loadatt48.m

In the figure above we can see that our fitness is increasing relatively fast in the beginning to later slowly start to converge. With this simulation of 4000 generations we can not say it has reached a steady point since it is increasing all the way but we can see that we are moving towards a specific range of values.

Results from simulation. (loadst70.m)

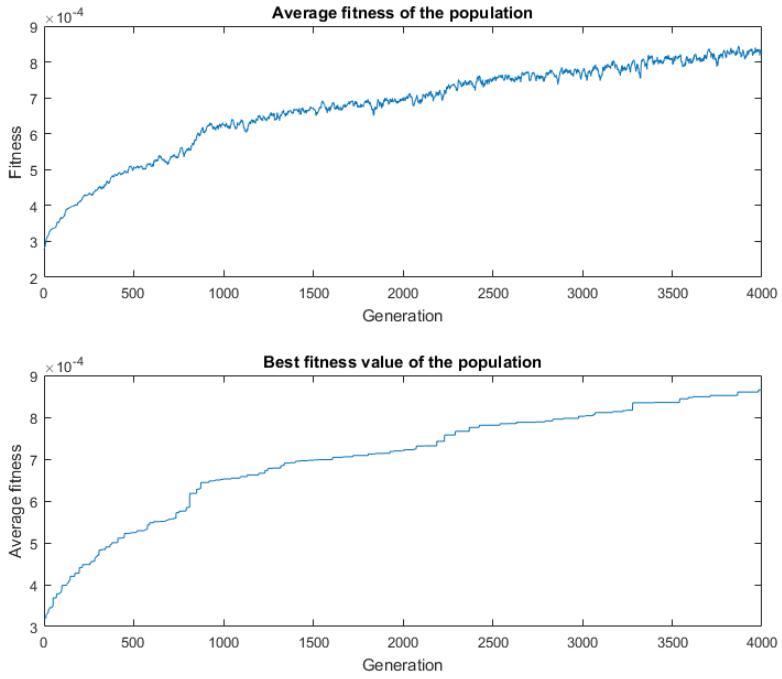

Figure 2: Simulation of 4000 generation with data from loadst70.m

The simulation above shows similar behaviour as the first simulation with a slight difference in convergence speed. I would say this is a wanted result since we keep the genetic variance in our population for a longer time decreasing chances of premature convergence.

Results from simulation. (loadgr96.m)

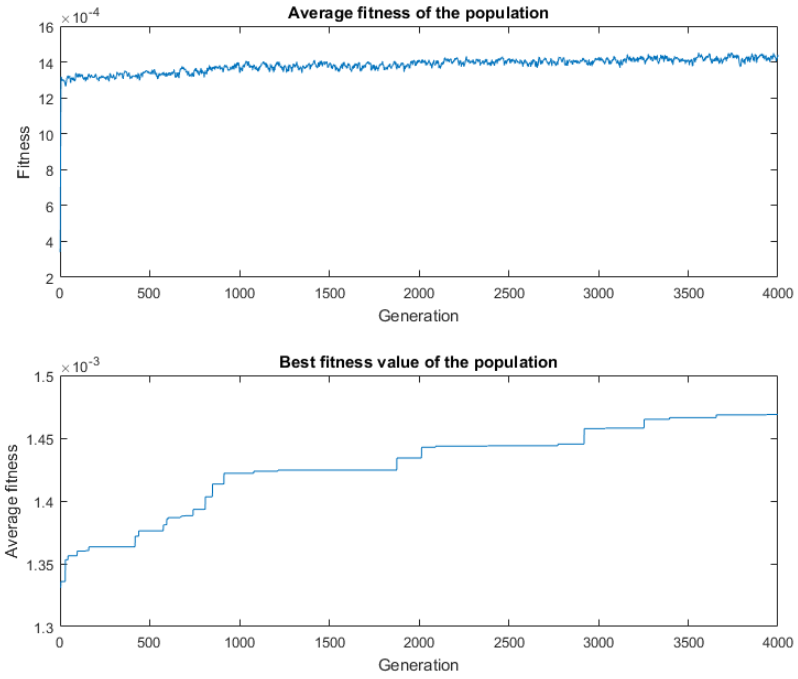

Figure 3: Simulation of 4000 generation with data from loadgr96.m

The last simulation suffer from premature convergence. This could be a result of loss of genetic variation and that our genetic operators are not finding offsprings superior to their parents. This is not a wanted result since we most likely are converging to fast against a value found early in the simulation. To increase genetic variation we could increase the population size and increasing the probability for mutation and cross over which could lead to finding better solutions along the simulation.

Task 2

Results from simulation. (loadatt48.m)

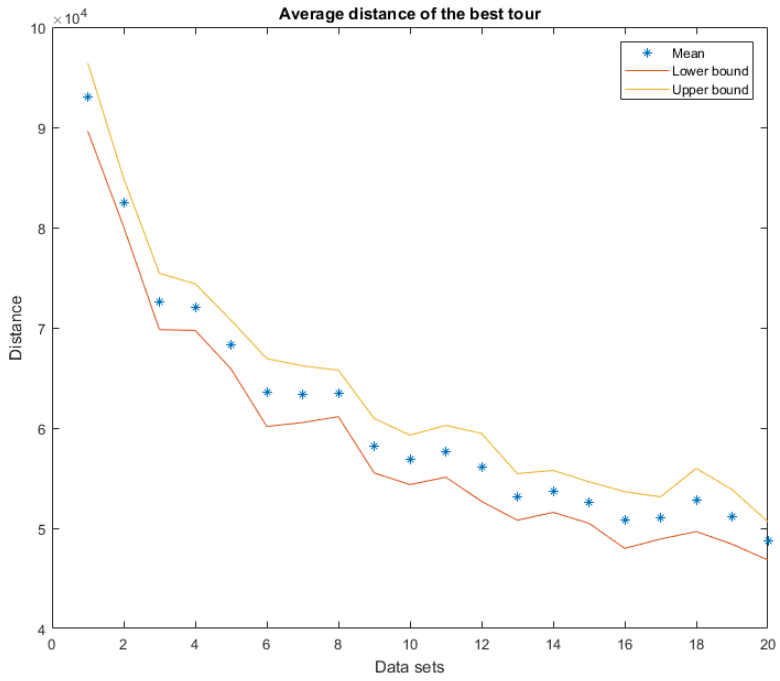

Figure 4: Data set results from simulations done with different number of generations.

Results from simulation. (loadst70.m)

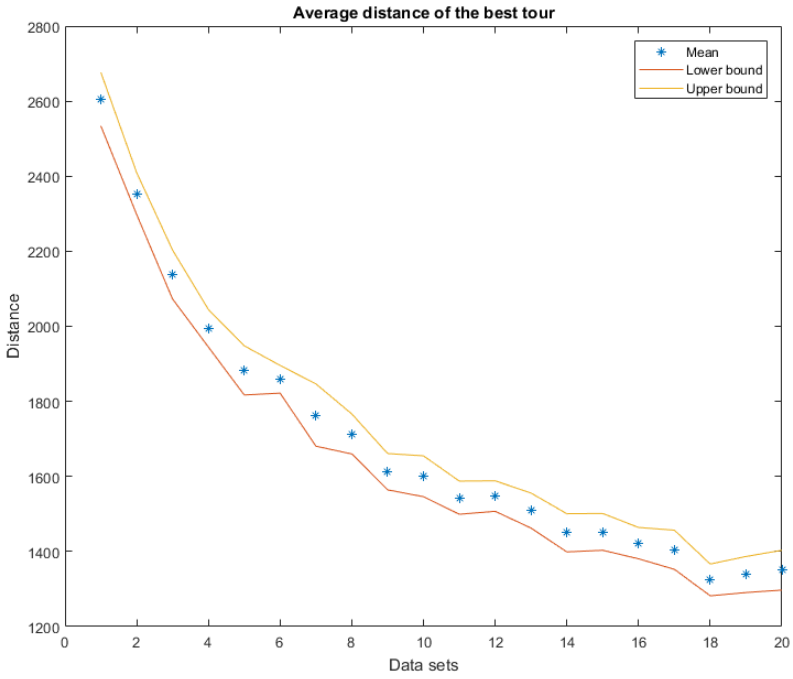

Figure 5: Data set results from simulations done with different number of generations.

Results from simulation. (loadgr96.m)

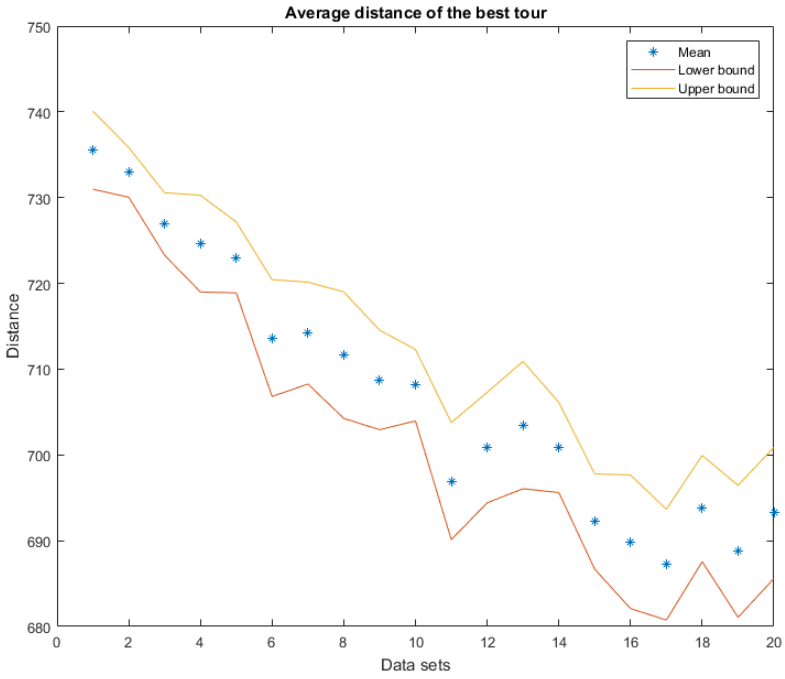

Figure 6: Data set results from simulations done with different number of generations.

The three different results differ in length of the confidence interval. The length of the confidence interval tells us how certain we can be about the results we have found and in which area the real mean is with a specific confidence, this time 95%. We do see more distinct convergence in the first two simulations and the confidence interval is tighter. In the last simulation we also do see a distinct increase in length of the confidence interval, which is a sign that we cannot trust our results even if we increase the number of generations since the results become more uncertain.

Code

1,2

tsp_ga.m

```
function varargout = tsp_ga(varargin)

% Initialize default configuration (do not change this part)
defaultConfig.xy       = 10*rand(50,2); % xy is the coordinate matrix
defaultConfig.dmat     = [];            % dmat is the distance matrix
defaultConfig.popSize  = 200;
defaultConfig.numGen   = 2000;
defaultConfig.crossProb = 0.25;
defaultConfig.mutProb  = 0.5;
defaultConfig.eliteFract = 0.02;

% Interpret user configuration inputs (do not change this part)
if ~nargin
    userConfig = struct();
elseif isstruct(varargin{1})
    userConfig = varargin{1};
else
    try
        userConfig = struct(varargin{:});
    catch
        error('Expected inputs are either a structure or parameter/value
            pairs');
    end
end

% Override default configuration with user inputs (do not change this
    part)
configStruct = get_config(defaultConfig,userConfig);

% Extract configuration
xy         = configStruct.xy; % xy is the coordinate matrix
dmat       = configStruct.dmat; % dmat is the distance matrix
popSize    = configStruct.popSize;
numGen     = configStruct.numGen;
crossProb  = defaultConfig.crossProb;
mutProb    = defaultConfig.mutProb;
eliteFract = defaultConfig.eliteFract;

if isempty(dmat)
    nPoints = size(xy,1);
    a = meshgrid(1:nPoints);
    dmat = reshape(sqrt(sum((xy(a,:)-xy(a',:)).^2,2)),nPoints,nPoints);
end
```

```
% Verify Inputs (do not change this part)
[N,dims] = size(xy);
[nr,nc] = size(dmat);
if N ~= nr || N ~= nc
    error('Invalid XY or DMAT inputs!')
end
n = N; % make sure you do not use this varaible n for other puposes
     (e.g. for loop iteration)



%%%%%%%%%%%%%%%% Initialize the Population
     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% You don't need to change this
     part%%%%%%%%%%%%%%%%%%%%%%%%
pop = zeros(popSize,n); % Matrix pop maintains the current population
pop(1,:) = (1:n);
for k = 2:popSize
    pop(k,:) = randperm(n);
end
%%%%%%%%%%%%%%%% End of Population Initialization
     %%%%%%%%%%%%%%%%%%%%%%%%



totalDist = zeros(1,popSize);
% totalDist is the vector of distances.Each element of this vector
     corresponds
% to the total distance (i.e. length) of a member of the population.

bestFitnessResults = zeros(1,numGen);
meanFitnessResults = zeros(1,numGen);

%% Starting GA iterations. In each iteration, a new generation is
     created %%%%%%
for iter = 1:numGen

    % Function calcToursDistances evaluates Each population member and
    % calculates the total distance of each member
    totalDist = calcToursDistances(pop, popSize, dmat, n);



    % Elite selection: you should use the information in matrix totalDist
    % to select a fraction eliteFract of the best members of the current
    % population pop. Keep these elite members in matrix elitePop.
    % Your elite selection code goes here:
    % ...
    % ...
```

```
% ...
fitness = 1./totalDist;
eliteSize = round(popSize*eliteFract);
eliteIndex = zeros(1, eliteSize);
% Finding index of elite members
for i=1:eliteSize
    [M,I] = max(fitness);
    eliteIndex(i) = I;
    fitness(I) = 0;
end
% Extracting elite members to elitePop
elitePop=zeros(eliteSize,n);
for i=1:eliteSize
    elitePop(i,1:end)=pop(eliteIndex(i),1:end);
end
%%%%%%% end of elite selection %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%



% Selection of new population: use totalDist to clacluate the fitness
% and the cummulative probability distribution of the current
    population
% pop. Then apply a roulette wheel selection to create a new
    population.
% Keep this new population in matrix newPop.
% Your roulette wheel selection code goes here:
% ...
% ...
% ...
% 1) Calculate the fitness value
fitness = 1./totalDist;
% 2) Find the total fitness of the population
F = sum(fitness);
% 3) Calculate the probability of a selection for each chromosome
p = fitness./F;
q = zeros(1,popSize);

% 4) Calculate a cumulative probability for each chromosome
for i=1:popSize
    temp = 0;
    for j=1:i
        temp = temp + p(j);
    end
    q(1,i) = temp;
end

% Selection process
newPop = zeros(popSize,n);
nbrOfC = 0;
while nbrOfC<=popSize
```

```
    temp = rand;
    if temp<q(1,1)
        nbrOfC = nbrOfC + 1;
        newPop(nbrOfC,1:end) = pop(1,1:end);
    else
        for i=2:popSize
            if temp>q(1,i-1) && temp<=q(1,i)
                nbrOfC = nbrOfC + 1;
                newPop(nbrOfC,1:end) = pop(i,1:end);
                break;
            end
        end
    end
end

%%%%%% end of roulette wheel selection
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Update distance vector totalDist according to the selected
    population
% newPop. Your code for updating totalDist goes here:
% ...
% ...
% ...
totalDist = calcToursDistances(newPop, popSize, dmat, n);
%%%%%% end of totalDist update
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Use the updated totalDist to calculate the new fitness values and
% cummulative probability distribution. Your code goes here:
% ...
% ...
% ...
% 1) Calculate the fitness value
fitness = 1./totalDist;
% 2) Find the total fitness of the population
F = sum(fitness);
% 3) Calculate the probability of a selection for each chromosome
p = fitness./F;
q = zeros(1,popSize);

% 4) Calculate a cumulative probability for each chromosome
for i=1:popSize
    temp = 0;
    for j=1:i
        temp = temp + p(j);
```

```
    end
    q(1,i) = temp;
end
%%%%% end of fitness and probabilty distribution update
     %%%%%%%%%%%%%%

% Cross-over operator: implement the cross-over procedure
% described in the home assignment. Use the cross-over
% probability crossProb to obtain the parents that should
% be selected purely random from the population newPop.
% Your code goes here:
% your cross-over code
% ...
% ...
% ...
% Selection of parents to crossover
nbrOfC = 0;
parentIndex = zeros(1,popSize);
for i=1:popSize
    r=rand;
    if r<crossProb
        nbrOfC = nbrOfC + 1;
        parentIndex(1,nbrOfC) = i;
    end
end

% Check if even number
if mod(nbrOfC,2)
    nbrOfC = nbrOfC - 1;
end

% Random parent crossover

while nbrOfC>0
    % Select parent 1
    i1 = randi([1, nbrOfC]);
    P1 = newPop(parentIndex(i1),1:end);
    parentIndex(i1)=parentIndex(nbrOfC);
    nbrOfC = nbrOfC - 1;
    % Select parent 2
    i2 = randi([1, nbrOfC]);
    P2 = newPop(parentIndex(i2),1:end);
    parentIndex(i2)=parentIndex(nbrOfC);
    nbrOfC = nbrOfC - 1;

    K=floor(0.3*size(P1,2));
    T1=P1;
```

```
T2=P2;
O=[];

while size(T1,2)~=0
    % Split Tl in two sub-tours T11 and T12 such that length of
        T11 = min {length of T1, K};
    length=min(size(T1,2),K);
    T11=T1(1,1:length);
    T12=T1(1,length+1:end);

    % Append T11 to O;
    O=[O T11];

    for i=1:size(T11,2)
        % Update Tl by removing from it the cities in T11
        j = 1;
        while 1
            if T1(1,j)==T11(1,i)
                T1(j)=[];
            end
            j = j + 1;
            if j>size(T1,2)
                break; % Since matrix is getting smaller
            end
        end

        % Update T2 by removing from it the cities in T11
        j = 1;
        while 1
            if T2(1,j)==T11(1,i)
                T2(j)=[];
            end
            j = j + 1;
            if j>size(T2,2)
                break; % Since matrix is getting smaller
            end
        end
    end
    X=T1;
    T1=T2;
    T2=X;
end
offspring = O;
%%%%%%%%%%%%% Mutation Operator
     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
r = rand();
if r <= mutProb

    routeInsertionPoints = sort(ceil(n*rand(1,2)));
    I = routeInsertionPoints(1);
```

```
        J = routeInsertionPoints(2);

        % 2-opt mutation (simply swaps two cities)
        offspring([I J]) = offspring([J I]);

        % now, you should replace one of the parents invloved in
        % cross-over with this mutated offspring, then update the
        % population newPop.
    end
    %%%%%%%%%% End of mutation operator
         %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

    r = randi([0, 1]);

    if r==1
        newPop(parentIndex(i1),1:end) = offspring(1,1:end);
    else
        newPop(parentIndex(i2),1:end) = offspring(1,1:end);
    end
end
%%%%%%% End of cross-over operator
     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Now, it is time to replace the worst members of newPop with the
     elite
% members you stored in matrix elitePop (see Elite selection in the
     begining
% of this iteration).
% Your code goes here:
% ...
% ...
% ...
% ...
totalDist = calcToursDistances(newPop, popSize, dmat, n);
worstIndex = zeros(1, eliteSize);
for i=1:eliteSize
    [M,I] = max(totalDist);
    worstIndex(i) = I;
    totalDist(I) = 0;
end
% Replacing worst members with elite members
for i=1:eliteSize
    newPop(worstIndex(i),1:end)=elitePop(i,1:end);
end

%%%%%%% End of elite replacement
     %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
    % Finally, the new population newPop should become the current
        population.
    pop = newPop;  % Uncomment this line when you finished all previous
    % steps

    totalDist = calcToursDistances(pop, popSize, dmat, n);
    [minDist,index] = min(totalDist);
    bestFitnessResults(1,iter) = 1/minDist;
    fitness = 1./totalDist;
    meanFitnessResults(1,iter) = mean(fitness);
end
%%%%%% End of GA ietartions
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


% Now, we find the best route in the last generation (you don't need to
% change this part). The best route is returned in optRoute, and its
% distance in minDist.
totalDist = calcToursDistances(pop, popSize, dmat, n);
[minDist,index] = min(totalDist);
optRoute = pop(index,:);

% Return Output (you don't need to change this part)
if nargout
    resultStruct = struct( ...
        'meanFitnessResults', meanFitnessResults, ...
        'bestFitnessResults', bestFitnessResults, ...
        'optRoute',   optRoute, ...
        'minDist',    minDist);

    varargout = {resultStruct};
end

end


% The following code is for configuation of user input (do not change
% this). Subfunction to override the default configuration with user
    inputs
function config = get_config(defaultConfig,userConfig)

% Initialize the configuration structure as the default
config = defaultConfig;

% Extract the field names of the default configuration structure
defaultFields = fieldnames(defaultConfig);

% Extract the field names of the user configuration structure
userFields = fieldnames(userConfig);
nUserFields = length(userFields);
```

```
% Override any default configuration fields with user values
for i = 1:nUserFields
    userField = userFields{i};
    isField = strcmpi(defaultFields,userField);
    if nnz(isField) == 1
        thisField = defaultFields{isField};
        config.(thisField) = userConfig.(userField);
    end
end

end
```

1

main_ga_task1.m

```
%   TSP_GA Traveling Salesman Problem (TSP) Genetic Algorithm (GA).
%   Finds a (near) optimal solution to the TSP by setting up a GA to
    search
%   for the shortest route (least distance for the salesman to travel to
%   each city exactly once and return to the starting city)
%
% Summary:
%     1. A single salesman travels to each of the cities and completes
    the
%        route by returning to the city he started from
%     2. Each city is visited by the salesman exactly once
%
% Input:
%     USERCONFIG (structure) with zero or more of the following fields:
%     - xy (float) is an Nx2 matrix of city locations, where N is the
    number of cities
%     - popSize (scalar integer) is the size of the population (should
    be divisible by 4)
%     - numGen (scalar integer) is the number of desired iterations for
    the algorithm to run
%     - mutProb (float) is the probability of mutation per individual
%     - crossProb (float) is the probability of cross-over
%     - eliteFract (float) is used for Elitism. It is the fraction of
    good members of the current
%     generation which should replace the weak members in the next
    generation.

% Input Notes:
%     1. Rather than passing in a structure containing these fields,
    any/all of
%        these inputs can be passed in as parameter/value pairs in any
```

```
    order instead.
%    2. Field/parameter names are case insensitive but must match
    exactly otherwise.
%
% Output:
%    RESULTSTRUCT (structure) with the following fields:
%        (in addition to a record of the algorithm configuration)
%    - OPTROUTE (integer array) is the best route found by the algorithm
%    - MINDIST (scalar float) is the cost of the best route

clc;
clear all;

% first, delete any old mat file from the workspace (you do not need to
    change this code)
if exist('cities.mat', 'file')==2
   delete('cities.mat');
end


%% Then, choose which data file you wish to experiment with.
% Below is the list of various data files with different number of
    cities:

% loadatt48();       % 48 cities
% loadst70();        % 70 cities
% loadgr96();        % 96 cities

% I have chosen a data file from the above list
 loadgr96();

%% prepare the distance matrix
load('cities.mat');
xy = cities';

% you should update the following code to obtain the average and 95%
% confidence interval for each configuration of numGen
meanFitness = zeros(1,400);
bestFitness = zeros(1,400);
for numGen = 4000

       userConfig = struct('xy', xy, 'popSize', 200, 'numGen', numGen,
           'crossProb', 0.25, 'mutProb', 0.5, 'eliteFract', 0.02);
       resultStruct = tsp_ga(userConfig);
       % the best tour found by GA
       % fprintf('\nBest tour found by GA:\n');
       % resultStruct.optRoute

       % the distance of the best tour
       %fprintf('\n Number of generations: %d \n Run number: %d \n The
```

```
            distance of the best tour = %d\n',numGen, runs,
            resultStruct.minDist);

end

% Implement your plotting here, using the average and confidence
    interval results:
% plots ...
figure
subplot(2,1,1)
plot(resultStruct.meanFitnessResults(1,:))
title('Average fitness of the population')
xlabel('Generation')
ylabel('Fitness')
subplot(2,1,2)
plot(resultStruct.bestFitnessResults(1,:));
title('Best fitness value of the population')
xlabel('Generation')
ylabel('Fitness')
```

2

main_ga_task2.m

```
%  TSP_GA Traveling Salesman Problem (TSP) Genetic Algorithm (GA).
%  Finds a (near) optimal solution to the TSP by setting up a GA to
    search
%  for the shortest route (least distance for the salesman to travel to
%  each city exactly once and return to the starting city)
%
% Summary:
%    1. A single salesman travels to each of the cities and completes
    the
%       route by returning to the city he started from
%    2. Each city is visited by the salesman exactly once
%
% Input:
%    USERCONFIG (structure) with zero or more of the following fields:
%    - xy (float) is an Nx2 matrix of city locations, where N is the
    number of cities
%    - popSize (scalar integer) is the size of the population (should
    be divisible by 4)
%    - numGen (scalar integer) is the number of desired iterations for
    the algorithm to run
%    - mutProb (float) is the probability of mutation per individual
%    - crossProb (float) is the probability of cross-over
%    - eliteFract (float) is used for Elitism. It is the fraction of
    good members of the current
```

```
%    generation which should replace the weak members in the next
    generation.

% Input Notes:
%    1. Rather than passing in a structure containing these fields,
    any/all of
%       these inputs can be passed in as parameter/value pairs in any
    order instead.
%    2. Field/parameter names are case insensitive but must match
    exactly otherwise.
%
% Output:
%    RESULTSTRUCT (structure) with the following fields:
%        (in addition to a record of the algorithm configuration)
%    - OPTROUTE (integer array) is the best route found by the algorithm
%    - MINDIST (scalar float) is the cost of the best route

clc;
clear all;

% first, delete any old mat file from the workspace (you do not need to
    change this code)
if exist('cities.mat', 'file')==2
   delete('cities.mat');
end


%% Then, choose which data file you wish to experiment with.
% Below is the list of various data files with different number of
    cities:

% loadatt48();       % 48 cities
% loadst70();        % 70 cities
% loadgr96();        % 96 cities

% I have chosen a data file from the above list
loadgr96();

%% prepare the distance matrix
load('cities.mat');
xy = cities';

% you should update the following code to obtain the average and 95%
% confidence interval for each configuration of numGen
data = zeros(20,15);
i = 1;
for numGen = 100:100:2000
   for runs = 1:15
       userConfig = struct('xy', xy, 'popSize', 200, 'numGen', numGen,
           'crossProb', 0.25, 'mutProb', 0.5, 'eliteFract', 0.02);
```

```
        resultStruct = tsp_ga(userConfig);
        data(i,runs) = resultStruct.minDist;
        % the best tour found by GA
        % fprintf('\nBest tour found by GA:\n');
        % resultStruct.optRoute

        % the distance of the best tour
        fprintf('\n Number of generations: %d \n Run number: %d \n The
            distance of the best tour = %d\n',numGen, runs,
            resultStruct.minDist);

    end
    i = i + 1;
end

% Implement your plotting here, using the average and confidence
    interval results:
% plots ...

% Calculating CIs
lower = zeros(1,20);
upper = zeros(1,20);
meanValue = zeros(1,20);
for i=1:20
    x = data(i,1:end);
    SEM = std(x)/sqrt(length(x));           % Standard Error
    ts = tinv([0.025 0.975],length(x)-1);   % T-Score
    CI = mean(x) + ts*SEM;                  % Confidence Intervals
    lower(1,i) = CI(1);
    upper(1,i) = CI(2);
    meanValue(1,i) = mean(x);
end

% Plot

plot(meanValue,'*');
hold on
plot(lower);
hold on
plot(upper);
legend('Mean','Lower bound','Upper bound')
title('Average distance of the best tour')
xlabel('Data sets')
ylabel('Distance')
```