Project Report
Genetic Algorithms and Evolutionary Computing [H02D1A]

Jasper Bau, Daan Seynaeve

January 6, 2016

1 Parameter Experiments

```
rondrit16.tsp
--
default settings:
3.5837

Cannot be optimal. Visibile inneffiencies in the tour. Can be optimized by changing only 2 edges. (s

run 1
Best solution was already there 50 generations ago

run2
not the case


minder generaties (100 default):
50: 3.81, 4.10, 4.01, 3.56, 3.81
(meer crossing edges)
(greater tour length)

25: 3.94, 4.17, 3.57, 4.2, 4.3


minder individuals (50 default):
25: 4.07, 3.55, 3.58, 3.64, 3.07
(less diversity)

meer indivduals:
200: 3.49, 3.45, 3.35, 3.36, 3.41
(consistent result, no crossing edges)

100 individuals, 50 gen:
```

```
3.66, 3.67, 3.50, 3.51, 3.54


No elites:
(No convergence, oscillating, random behaviour)
bad results
more generations --> no influence
more individuals --> less extreme oscillation. Average fitness levels out. Seems to be stuck at subo

More individuals warrants more elite?

High amount of elites -> stuck at specific solutions for a long time. Population converges to a sing

Enabling loop detection -> much better performance. Faster saturation + convergence.

Prob. for mutation or crossover must be high enough for anything to happen...

If no mutation -> algo can become 'stuck'.

no mutation + low crossover prob -> algo comes to an early stop.

crossover increasingly important for larger problem instances?
```

2 Path Representation

Some operators: A.3 and A.1

3 Optional Task

4 Benchmark Performance

4.1 rbx711

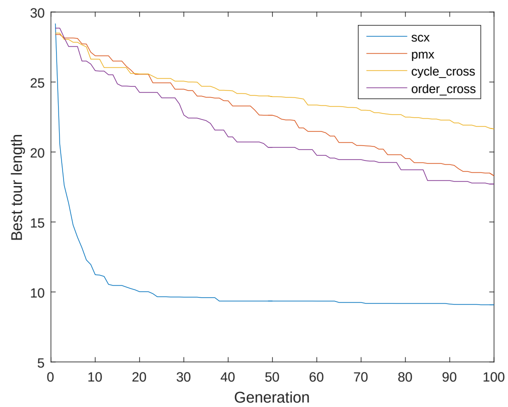

Figure 1: Comparison of path representation crossover operators on the rondrit127 dataset (50 individuals, 95% crossover, 20% inversion mutation, no local loop optimization)

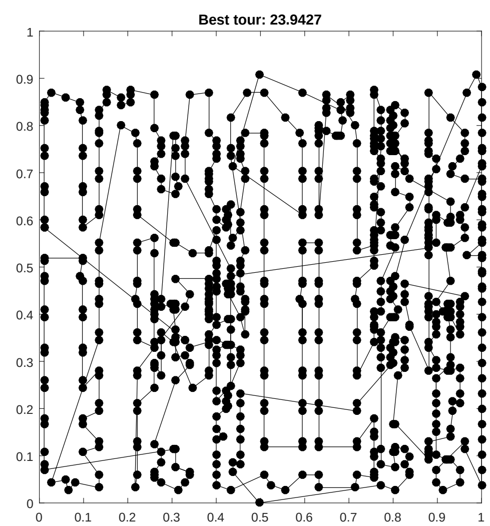

Figure 2: rbx711 benchmark: result of 100 generations with a population size of 100 using SCX crossover (95%) with inversion mutation (35%).

Appendix A Code

A.1 scx.m

```
% Sequential Constructive crossover for TSP (Zakir H. Ahmed, 2010)
% this crossover assumes that the path representation is used to represent
% TSP tours
%
% Daan Seynaeve 2015
%
%
% Syntax:  NewChrom = scx(OldChrom,Cost)
%
% Input parameters:
%    OldChrom  - Matrix containing the chromosomes of the old
%                population. Each line corresponds to one individual
%                (in any form, not necessarily real values).
%
%    Cost      - Cost matrix. Cost(i,j) corresponds to the cost between
%                the node i and the node j
%
% 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 = scx(oldChrom,Cost,x_probability)
    if nargin < 3
        x_probability = 1;
    end
    if rand<x_probability
        n = size(oldChrom,2);
        child1 = zeros(1,n);
        legit = ones(1,n);

        parent1 = oldChrom(1,:);
        parent2 = oldChrom(2,:);

        nci = 1;
        child1(1) = 1;
        legit(1) = 0;
        while nci < n
            p = child1(nci);
            nci = nci + 1;

            alfa = 0;
            beta = 0;

            for i = 1:n-1
                if parent1(i) == p && legit(parent1(i+1))
                    alfa = parent1(i + 1);
                end
                if parent2(i) == p && legit(parent2(i+1))
                    beta = parent2(i + 1);
                end
            end

            if alfa == 0 || beta == 0
                % find a legit k
                k = 2;
                while not(legit(k))
                    k = k + 1;
                end

                if beta == 0 && alfa == 0
                    beta = k;
                    alfa = k;
                elseif beta == 0
                    beta = k;
                else
                    alfa = k;
                end
```

```
            end
            if Cost(p,alfa) < Cost(p,beta)
                child1(nci) = alfa;
            else
                child1(nci) = beta;
            end
            legit(child1(nci)) = 0;
        end
        NewChrom = [parent1; child1];
    else
        NewChrom = oldChrom;
    end

end
```

A.2 pmx.m

```
% Partially mapped crossover for the TSP problem
% Crossover operator for the path representation

function NewChrom = partially_mapped_cross(OldChrom, x_probability)
    if nargin<2
        x_probability = 1;
    end
    if rand<x_probability
        % select 2 random crossover points and take the segments between those
        % points in the parents as offspring
        NewChrom = zeros(2,length(OldChrom(1,:)));
        cross_over_points = randi([1,length(OldChrom(1,:))],1,2);
        if cross_over_points(1) > cross_over_points(2)
            cross_over_points = flip(cross_over_points);
        end
        offspring1 = OldChrom(1,cross_over_points(1):cross_over_points(2));
        offspring2 = OldChrom(2,cross_over_points(1):cross_over_points(2));
        NewChrom(1,cross_over_points(1):cross_over_points(2)) = offspring1;
        NewChrom(2,cross_over_points(1):cross_over_points(2)) = offspring2;

        % Find the elements of the offspring of the second parent that have not been copied yet
        elements_to_copy = [];
        for i=1:length(offspring2)
            if isempty(find(offspring1==offspring2(i)))
                elements_to_copy = [elements_to_copy offspring2(i)];
            end
        end
        % Search for the corresponding element in the offspring of the first
        % parent for eacht element in elements_to_copy
        % Search for the index of that element in the second parent and put the
        % element i of elements_to_copy in that position in child 1
        % if that position in child one is already taken repeat this procedure
        for i=1:length(elements_to_copy)
            index = find(offspring2 == elements_to_copy(i));
            element_copied_instead = offspring1(index);
            index_parent2 = find(OldChrom(2,:)==element_copied_instead);
            while NewChrom(1,index_parent2) ~= 0
                element_copied_instead = NewChrom(1,index_parent2);
                index_parent2 = find(OldChrom(2,:)==element_copied_instead);
            end
            NewChrom(1,index_parent2) = elements_to_copy(i);
        end
        % copy the remaining elements to child 1
        positions_still_zero = find(NewChrom(1,:)==0);
        for i=1:length(positions_still_zero)
            NewChrom(1,positions_still_zero(i)) = OldChrom(2,positions_still_zero(i));
        end

        % analogous for the second child
        elements_to_copy = [];
        for i=1:length(offspring1)
            if isempty(find(offspring2==offspring1(i)))
                elements_to_copy = [elements_to_copy offspring1(i)];
            end
```

```
        end
        for i=1:length(elements_to_copy)
            index = find(offspring1 == elements_to_copy(i));
            element_copied_instead = offspring2(index);
            index_parent1 = find(OldChrom(1,:)==element_copied_instead);
            while NewChrom(2,index_parent1) ~= 0
                element_copied_instead = NewChrom(2,index_parent1);
                index_parent1 = find(OldChrom(1,:)==element_copied_instead);
            end
            NewChrom(2,index_parent1) = elements_to_copy(i);
        end
        positions_still_zero = find(NewChrom(2,:)==0);
        for i=1:length(positions_still_zero)
            NewChrom(2,positions_still_zero(i)) = OldChrom(1,positions_still_zero(i));
        end
    else
        NewChrom = OldChrom;
    end
end
```

A.3 cycle_cross.m

```
% Cycle crossover for the TSP problem
% Crossover operator for the path representation

function NewChrom = cycle_cross(OldChrom, x_probability)
    if nargin<2
        x_probability = 1;
    end
    if rand<x_probability
        parent1 = OldChrom(1,:);
        parent2 = OldChrom(2,:);

        % construct a cycle and use it as offspring
        start_cycle = parent1(1);
        next_in_cycle = parent2(1);
        child1 = [start_cycle zeros(1,length(parent1)-1)];
        child2 = [next_in_cycle zeros(1,length(parent2)-1)];

        while next_in_cycle ~= parent1(1)
            index_other_parent = find(parent1 == next_in_cycle);
            child1(index_other_parent) = next_in_cycle;
            next_in_cycle = parent2(index_other_parent);
            child2(index_other_parent) = next_in_cycle;
        end

        % exchange the elements that are not copied yet
        positions_zero = find(child1==0);
        for i=positions_zero
            child1(i) = parent2(i);
            child2(i) = parent1(i);
        end

        NewChrom(1,:) = child1;
        NewChrom(2,:) = child2;
    else
        NewChrom = OldChrom;
    end
end
```

A.4 order_cross.m

```
% Order crossover for the TSP problem
% Crossover operator for the path representation

function NewChrom = order_cross(OldChrom, x_probability)
    if nargin<2
        x_probability = 1;
    end
    if rand<x_probability
        % select 2 random crossover points and take the segments between those
```

```
% points in the parents as offspring
NewChrom = zeros(2,length(OldChrom(1,:)));
cross_over_points = randi([1,length(OldChrom(1,:))],1,2);
if cross_over_points(1) > cross_over_points(2)
    cross_over_points = flip(cross_over_points);
end
offspring1 = OldChrom(1,cross_over_points(1):cross_over_points(2));
offspring2 = OldChrom(2,cross_over_points(1):cross_over_points(2));
NewChrom(1,cross_over_points(1):cross_over_points(2)) = offspring1;
NewChrom(2,cross_over_points(1):cross_over_points(2)) = offspring2;

% Search for the index of the elements of the second parent that have
% not been copied yet
indexes_elements = [];
for i=1:length(OldChrom(2,:))
    if isempty(find(offspring1==OldChrom(2,i)))
        indexes_elements = [indexes_elements i];
    end
end

% Order the index matrix in the order to copy
% Not necessary if the second crossover point is equal to the length of
% the second parent

% Determine the index of the element to copy first
% Order indexes_elements
if ~isempty(indexes_elements)
    if (cross_over_points(2)~=length(OldChrom(2,:))) ...
            && (max(indexes_elements)>cross_over_points(2))
        start_index = min(indexes_elements(...
            find((indexes_elements>cross_over_points(2))==1)));
        indexes_elements = circshift(indexes_elements,...
            [0,(length(indexes_elements)+1)-find(indexes_elements==start_index)]);
    end
    % Copy the elements to child one in the order determined of
    % indexes_elements
    if (cross_over_points(2)~=length(OldChrom(2,:)))
        counter = cross_over_points(2)+1;
        for i=indexes_elements
            if counter == length(OldChrom(2,:))
                NewChrom(1,counter) = OldChrom(2,i);
                counter = 1;
            else
                NewChrom(1,counter) = OldChrom(2,i);
                counter = counter+1;
            end
        end
    else
        for i=1:cross_over_points(1)-1
            NewChrom(1,i) = OldChrom(2,indexes_elements(i));
        end
    end
end

% analogous for the second child

indexes_elements = [];
for i=1:length(OldChrom(1,:))
    if isempty(find(offspring2==OldChrom(1,i)))
        indexes_elements = [indexes_elements i];
    end
end

if ~isempty(indexes_elements)
    if (cross_over_points(2)~=length(OldChrom(1,:))) && ...
            (max(indexes_elements)>cross_over_points(2))
        start_index = min(indexes_elements(...
            find((indexes_elements>cross_over_points(2))==1)));
        indexes_elements = circshift(indexes_elements,...
            [0,(length(indexes_elements)+1)-find(indexes_elements==start_index)]);
    end
```

```
            if (cross_over_points(2)~=length(OldChrom(1,:)))
                counter = cross_over_points(2)+1;
                for i=indexes_elements
                    if counter == length(OldChrom(1,:))
                        NewChrom(2,counter) = OldChrom(1,i);
                        counter = 1;
                    else
                        NewChrom(2,counter) = OldChrom(1,i);
                        counter = counter+1;
                    end
                end
            else
                for i=1:cross_over_points(1)-1
                    NewChrom(2,i) = OldChrom(1,indexes_elements(i));
                end
            end
        end
    else
        NewChrom = OldChrom;
    end
end
```

A.5 run_ga_adapted.m

```
% Main algorithm
%
% Jasper Bau, Daan Seynaeve 2016
%
% parameters:
%
% x, y: coordinates of the cities
% REP: representation to use ('path' or 'adj')
% 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
% ah1, ah2, ah3: axes handles to visualise tsp
%
% returns: best-, mean- and worst-fitness w.r.t. gen
%
function [best,mean_fits,worst] = run_ga_adapted(...
    x, y, REP, NIND, MAXGEN, NVAR, ELITIST, STOP_PERCENTAGE, ...
    PR_CROSS, PR_MUT, CROSSOVER, MUTATION, LOCALLOOP, ...
    ah1, ah2, ah3)

    GGAP = 1 - ELITIST;
    stopN=ceil(STOP_PERCENTAGE*NIND);
    mean_fits=zeros(1,MAXGEN+1);
    worst=zeros(1,MAXGEN+1);
    best=zeros(1,MAXGEN);

    % distance matrix
    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);
    if strcmp(REP,'path');
        for row=1:NIND
            Chrom(row,:)=path2adj(randperm(NVAR));
        end
        ObjV = tspfun_path(Chrom,Dist);
    else
        for row=1:NIND
```

```
        Chrom(row,:)=randperm(NVAR);
        end
        ObjV = tspfun(Chrom,Dist);
    end

    % generational loop
    gen = 0;
    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

        pathData = Chrom(t,:);
        if strcmp(REP,'adj')
            pathData = adj2path(Chrom(t,:));
        end
        visualizeTSP(x,y,pathData, minimum, ah1, gen, best, mean_fits, worst, ah2, ObjV, NIND, ah3);

        % stop criterium
        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);

        if strcmp(REP,'path') % path representation
            % recombine individuals (crossover)
            SelCh = recombin_path(CROSSOVER,SelCh,PR_CROSS,Dist);
            SelCh = mutateTSP_path(MUTATION,SelCh,PR_MUT);

            %evaluate offspring, call objective function
            ObjVSel = tspfun_path(SelCh,Dist);

            %reinsert offspring into population
            [Chrom, ObjV]=reins(Chrom,SelCh,1,1,ObjV,ObjVSel);

            Chrom = tsp_ImprovePopulation_path(NIND, NVAR, Chrom,LOCALLOOP,Dist);
        else % adjacency representation
            % recombine individuals (crossover)
            SelCh = recombin(CROSSOVER,SelCh,PR_CROSS);
            SelCh = mutateTSP('inversion',SelCh,PR_MUT);

            %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);
        end
        %increment generation counter
        gen=gen+1;
    end
end
```