Genetic Algorithms and Evolutionary Computing Project

Jasper Hawinkel & Matthias Baeten

11 january 2016

Contents

1 Answer Question 2

Write the answer for question 2 in this file

Build the report by building the file `../build/build_report.tex`

Put the figures in `../figures/`

2 Path Representation

In this section an adjusted version of the existing genetic algorithm will be studied. The tours in the Traveling Salesman Problem will not be represented anymore by the adjacency representation. The path representation will be used instead. The ith element of the path representation denotes the ith city visited. This representation needs appropriate recombination and mutation operators. We chose to use the order crossover operator as recombination operator. For mutation we chose to use the inversion operator.

The order crossover operator recombines the genes of two parents to produce two children. To produce the first offspring a randomly chosen segment of the first parent is copied into the offspring. Secondly information about the relative order of the second parent is used to make the representation of the first offspring complete. The second offspring is created in an analogous manner, with the parent roles reversed. The working process of the order crossover operator is shown below:

1. Choose two crossover points at random, and copy the segment between them from the first parent into the first offspring.
2. Starting from the second crossover point in the second parent, and copy the remaining unused numbers into the first offspring in the order that they appear in the second parent.
3. Create the second offspring in an analogous manner, with the parent roles reversed.

The used mutation operator is the inversion mutation operator. This operator works by randomly selecting two positions in the chromosome and reversing the order in which the values appear between those positions. For more information about these operators we refer to [1]. The implementation of the order crossover operator is shown in the appendices. The code for evaluating the fitness of candidate solutions in path representation is also shown in the appendices. The path representation and the inversion mutation operator were already implemented in the template Matlab program for the TSP.

Now we will perform some parameter tuning to identify proper combinations of the parameters. Remark that genetic algorithms are stochastic. That is why we can not draw any conclusions about the quality of the genetic algorithm from a single run. Therefore multiple runs will be used to estimate the quality of the genetic algorithm for a certain set of algorithm parameters. For every run the computation time and the best candidate solution found are stored. The computations times and the fitness values of the most optimal solutions found are then averaged over the amount of runs. The results of the parameter tuning process are shown in table 1.

We started the parameter tuning process by using the default parameters used in the template program. These default parameters are shown in the second row of table 1. Next we studied every parameter separately by only adjusting the value of one parameters and using the default values for the other parameters. In that way we can find the optimal value for each parameter separately. At the end we then combine the optimal parameter values and hopefully this will give us the optimal set of parameters. As we already expected, increasing the population size and the number of generations improves the results. The disadvantage is that the computation time (and probably the amount of memory needed) increases also. It is clear that here some sort of trade-off has to be made between solution quality and CPU time (and memory needed). The optimal value for the probability of mutation seems to lie around 8% and the optimal value for the probability of recombination seems to lie around 80%. The optimal proportion of elite seems to lie around 15%. As we expected the results with loop detection are much better than without. With loop detection no significantly more CPU time is needed than without. So if loop detection is implemented, you should always use it.

In the four last columns of table 1 the optimal values for the mutation and crossover probability and elite proportion are used. The combined optimal values indeed result in very good results. It seems like it is more optimal to take the number of individuals equal

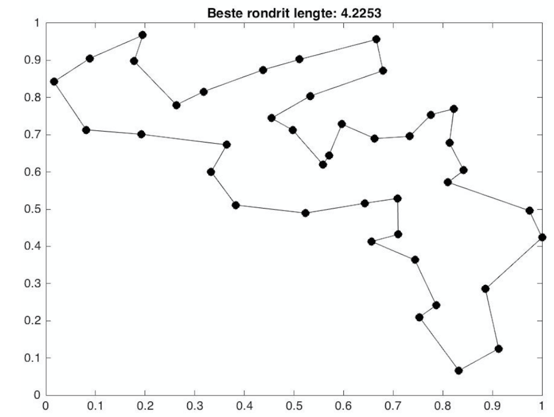
#IND	#GEN	PR. MUT	PR. CROS	ELITE	LP DET	DIST.	TIME [sec]
50	**100**	**5 %**	**95 %**	**5 %**	**OFF**	**10.99**	**10.80**
100	df	df	df	df	df	**9.962**	**12.56**
200	df	df	df	df	df	**9.425**	**15.09**
500	df	df	df	df	df	**8.926**	**25.18**
df	**200**	df	df	df	df	**8.919**	**21.83**
df	**500**	df	df	df	df	**6.721**	**52.53**
df	df	**0 %**	df	df	df	**10.95**	**10.75**
df	df	**8 %**	df	df	df	**10.76**	**10.76**
df	df	**15 %**	df	df	df	**11.21**	**11.08**
df	df	df	**70 %**	df	df	**10.66**	**9.499**
df	df	df	**80 %**	df	df	**10.54**	**10.41**
df	df	df	**90 %**	df	df	**10.70**	**10.74**
df	df	df	**100 %**	df	df	**10.86**	**10.82**
df	df	df	df	**0 %**	df	**15.61**	**11.01**
df	df	df	df	**10 %**	df	**10.17**	**9.855**
df	df	df	df	**15 %**	df	**9.954**	**9.787**
df	df	df	df	**20 %**	df	**10.14**	**9.520**
df	df	df	df	**25 %**	df	**10.38**	**9.661**
df	df	df	df	df	**ON**	**7.020**	**10.48**
100	**200**	**8 %**	**80 %**	**15 %**	**OFF**	**7.143**	**22.82**
200	**200**	**8 %**	**80 %**	**15 %**	**OFF**	**6.394**	**28.01**
100	**200**	**8 %**	**80 %**	**15 %**	**ON**	**5.357**	**23.09**
200	**200**	**8 %**	**80 %**	**15 %**	**ON**	**4.932**	**28.62**

Table 1: Table with the results of some parameter tuning of the genetic algorithm. A path representation is used with order crossover and inversion mutation. For every set of parameters 10 runs are calculated. The template TSP problem with 67 cities is used. The second last column contains the averages of the most optimal fitness values (minimal tour distance) found. The last column contains the averages of the computation time of a run. `df` stands for 'default value'. In the boxes with a `df` statement the default value from the first row is used. #IND = #INDIVIDUALS, #GEN = #GENERATIONS, PR. MUT = MUTATION PROBABILITY, PR. CROS = CROSSOVER PROBABILITY, LP DET = LOOP DETECTION, DIST = OPTIMAL DISTANCE.

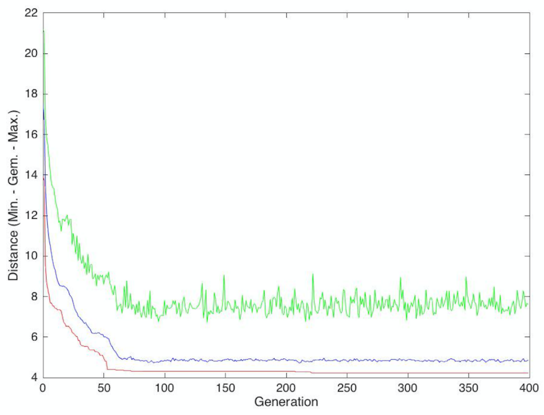
to the number of generators instead of only half. Increasing the number of individuals does not double the amount of computation time in contrast to the number of generations.

3 Benchmark Problems

In this section we will test the performance of our designed algorithm (path representation) using some larger benchmark problems. Again we tested the parameter values in table 1. And we came to the conclusion that the optimal parameter values found in table 1 are also the optimal parameter values for larger benchmark problems. We executed the tests on the benchmark problem containing 380 cities (`bcl380.tsp`). It is certainly not



(a) The most optimal tour found for the TSP problem. The distance of the tour is 4.2253.



(b) The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

Figure 1: The benchmark TSP problem `belgiumtour.tsp` with 41 cities is solved here with path representation, order crossover and inversion mutation. The settings of the GA were #IND = 400, #GEN = 400, PR. MUT = 8%, PR. CROS = 80%, , ELITE = 15%, LP DET = ON.

that logical that the parameter values that are optimal for small size problems are also the optimal values for larger size problems. Now that we have found the optimal parameter settings for our genetic algorithm, we can use the algorithm to solve some large benchmark problems. In figure 1 you can see the results for the Belgium tour benchmark problem. It seems that the GA found the optimal solution very quickly.

We also did a performance comparison of the original GA (adjacency representation) with the new designed GA (path representation). For both algorithms their optimal set of parameter values are used. In figure 2 and 3 the results are shown with loop detection switched off. We see that the new designed GA finds a shorter optimal tour than the original GA. The disadvantage of the new designed GA is that it takes more computation time than the original GA. In figure 4 and 5 the same experiment is done with loop detection switched on. Now both methods obtain approximately the same tour quality but the new designed GA takes again more CPU time. Notice that with the original GA in figure 4 the fitness values are decreasing very fast at the beginning and after that they decrease rather slow.If we compare this behavior with the new GA in figure 4 we see that with the new GA the fitness value decrease more gradually over the generation process.

It seems that with loop detection switched on, the original GA and the new GA obtain the same tour quality, but the new GA needs more computation time . Without loop detection the new GA gives better results, but it takes more CPU time. Remark that with the original GA that the fitness value of the worst solution in the population set is always far away from the fitness values of the best and average solution. With the new designed GA these three values lie much closer together.

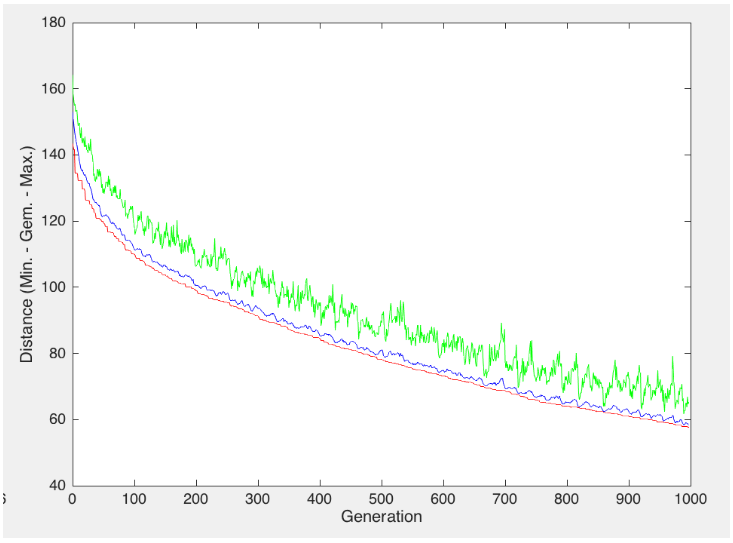


Figure 2: The benchmark TSP problem `belgiumtour.tsp` with 380 cities is solved here with **adjacency representation (original GA)**. The optimal candidate solution is far away from the best solution. **The optimal tour distance found is 57.14 and the computation time needed was around 209 seconds**. **The optimal settings of the original GA were used**, namely PR. MUT = 10%, PR. CROS = 50% , ELITE = 10%. 200 individuals and 1000 generations are used and **loop detection is switched off**. The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

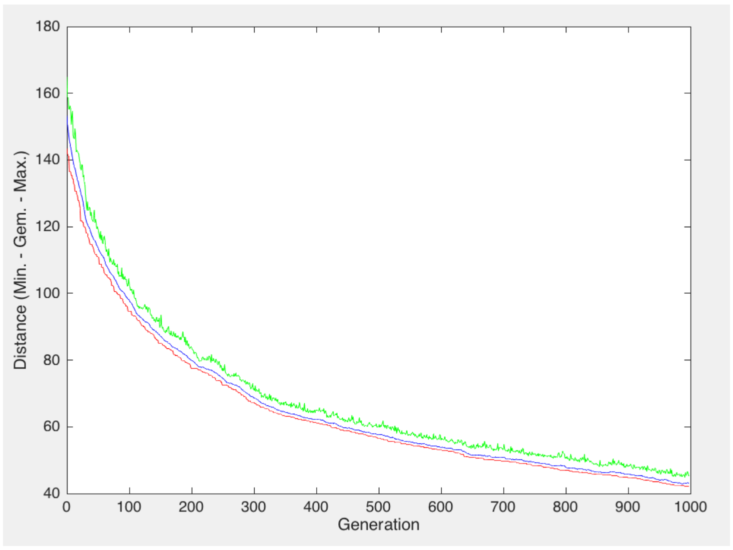


Figure 3: The benchmark TSP problem `belgiumtour.tsp` with 380 cities is solved here with **path representation (new designed GA)**. The optimal candidate solution is far away from the best solution. **The optimal tour distance found is 42.1 and the computation time needed was around 357 seconds**. **The optimal settings of the new designed GA were used**, namely PR. MUT = 8%, PR. CROS = 80% , ELITE = 15%. 200 individuals and 1000 generations are used and **loop detection is switched off**. The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

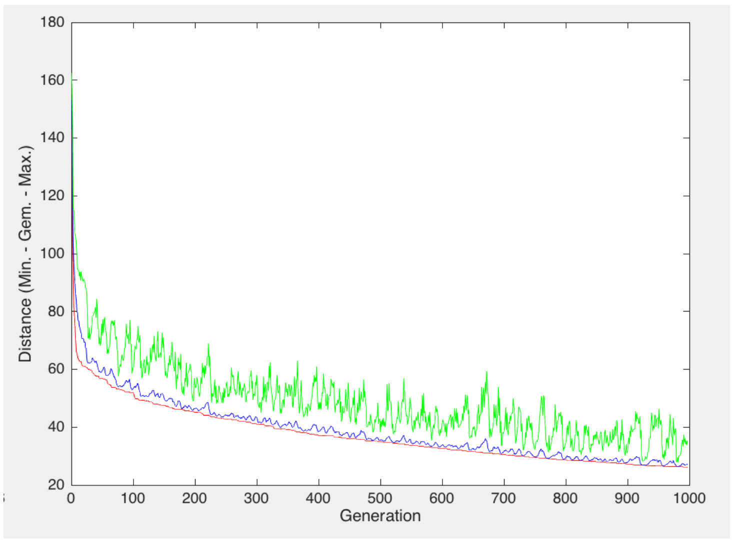


Figure 4: The benchmark TSP problem `belgiumtour.tsp` with 380 cities is solved here with **adjacency representation (original GA)**. The optimal candidate solution is far away from the best solution. **The optimal tour distance found is 26.2 and the computation time needed was around 226 seconds**. **The optimal settings of the original GA were used**, namely PR. MUT = 10%, PR. CROS = 50% , ELITE = 10%. 200 individuals and 1000 generations are used and **loop detection is switched on**. The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

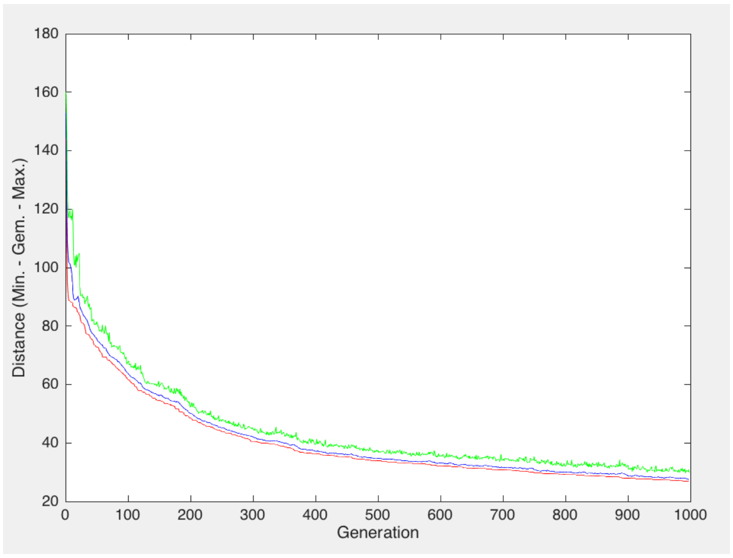


Figure 5: The benchmark TSP problem `belgiumtour.tsp` with 380 cities is solved here with **path representation (new designed GA)**. The optimal candidate solution is far away from the best solution. **The optimal tour distance found is 26.8 and the computation time needed was around 387 seconds**. **The optimal settings of the new designed GA were used**, namely PR. MUT = 8%, PR. CROS = 80% , ELITE = 15%. 200 individuals and 1000 generations are used and **loop detection is switched on**. The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

4 Answer Optional Question

Write the answer for the optional question in this file

References

[1] A. E. Eiben, J.E. Smith, *Introduction to Evolutionary Computing*, Second Edition.

Appendices

In the appendices you can find a small part of the code that is used to obtain the results discussed in this report. To make sure that the appendix is not extremely long, only the code is shown that is completely written by ourselves.

Matlab Code

Implementation Order Crossover

This function recombines two parents to create two children with the use of Order Crossover.

```
function Offspring = order_crossover(Parents)

% INPUT: -Parents: 2xn matrix containing the two parents
    representation
% OUTPUT: -Offspring: 2xn matrix containing the two offspring
    representation's

n = size(Parents,2);
rn1 = randi(n);
delta = randi(n-1); % size of copied piece
Offspring = zeros(2,n);

j = rn1;
Offspring_copy = zeros(2,delta);
idx = 1;

while delta >= 1
    Offspring(:, j) = Parents(:,j);
    Offspring_copy(:,idx) = Offspring(:, j);
    if j == n
        j = 1;
    else
        j = j+1;
    end
    delta = delta - 1;
    idx = idx+1;
end

offspring1_index = 1;
offspring2_index = 1;
while j ~= rn1
    % check for position j
    cand1 = Parents(2, offspring1_index);
    cand2 = Parents(1, offspring2_index);
    while (ismember(cand1,Offspring_copy(1,:)))
        offspring1_index=offspring1_index+1;
        cand1 = Parents(2, offspring1_index);
    end
    while (ismember(cand2,Offspring_copy(2,:)))
        offspring2_index=offspring2_index+1;
        cand2 = Parents(1, offspring2_index);
    end
    Offspring(1, j) = cand1;
    Offspring(2, j) = cand2;
    offspring1_index = offspring1_index +1;
    offspring2_index = offspring2_index +1;
    if j == n
        j = 1;
    else
        j = j+1;
    end
end

end
```

Implementation fitness function for path representation

This function compute the fitness values corresponding to the candidate solutions in path representation.

```

function ObjVal = tspfun_path(Phen, Dist)
%   Implementation of the TSP fitness function
%   Phen contains the phenocode of the matrix coded in
%   path representation.
%   Phen is a matrix of size NIND x NVAR with at every row
%   the path representation of a tour.
%   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)
```

```

size_Dist = size(Dist);
ObjVal = Dist( sub2ind(size_Dist , Phen(:,end), Phen(:,1)) );
for t = 1:size(Phen,2)-1
    ObjVal=ObjVal + Dist( sub2ind(size_Dist , Phen(:,t), Phen(:,t
       +1)) );
end

end
```