Genetic Algorithms and Evolutionary Computing Project

Jasper Hawinkel & Matthias Baeten

11 january 2016

Contents

1 Template TSP algorithm

This section discusses the parameter tuning and experimentation with the provided Matlab template. The results gained with this algorithm are used in the following sections to compare them with the results from path representation.

1.1 Parameter tuning

The template uses the adjecency representation with crossover and inversion mutation. The parent selection mechanism is rank selection combined with elitism as survivor selection. Table 1 shows the results of the performed parameter tuning on this algorithm. To produce these result the genetic algorithm was performed 10 times with each parameter setting on the *rondrit067.tsp* file. The values shown in the last 2 columns (i.e. the best tour and the running time) are the averages of the results from those 10 runs. We started out with some standard parameters for the algorithm and then one by one tuned each parameter to find a good value. Then we combined the best results for each parameter.

After some experimentation with this combination we settled on the parameters shown in the 3 to last row, with 20% mutation chance, 50% recombination and 10% elitism.

1.2 Results

There are several interesting results that can be learned from table 1. Firstly, increasing the population only seams to have a minor impact on the execution time while there is significant benefit to the best tour found by the algorithm. For this particular problem, a population of 200-300 seems to be a reasonable trade-off between maximum fitness and execution time.

Secondly the chance of recombination seems to be a very important factor for the genetic algorithm, with a good choice of parameter resulting not only in better fitness evolution but also in reduced execution time compared to our initial value. The big reduction in execution time (almost 25%) when going from 85% recombination chance to 50% indicates that a large part of the execution time is spent on recombination.

Thirdly, the parameters for mutation and elitism seem to have only a moderate impact on the performance of the genetic algorithm.

Lastly we tested the impact of loop detection on the genetic algorithm. This method removes loops of maximum 3 edges from all paths, increasing their fitness. From the results, it is clear that this method greatly improves maximum fitness with only a very small increase in execution time. The biggest downside to loop-detection is that this is a very problem specific method, limiting the range of problems the genetic algorithm can be applied to.

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.

#IND	**#GEN**	**PR. MUT**	**PR. CROS**	**ELITE**	**LP DET**	**DIST.**	**TIME [sec]**
100	100	10%	85%	5%	OFF	14.21	10.91
150	100	10%	85%	5%	OFF	13.75	11.4
200	100	10%	85%	5%	OFF	13.23	12.65
300	100	10%	85%	5%	OFF	12.51	14.95
200	150	10%	85%	5%	OFF	12.25	20.03
200	200	10%	85%	5%	OFF	11.57	26.6
200	150	20%	85%	5%	OFF	12.33	21.95
200	150	30%	85%	5%	OFF	12.22	22.25
200	150	10%	80%	5%	OFF	11.63	18.64
200	150	10%	70%	5%	OFF	10.53	17.63
200	150	10%	60%	5%	OFF	9.49	17.39
200	150	10%	50%	5%	OFF	8.98	16.43
200	150	10%	85%	5%	OFF	9.1	17.46
200	150	10%	85%	10%	OFF	8.64	15.79
200	150	10%	85%	15%	OFF	8.85	16.04
200	200	20%	50%	5%	OFF	7.89	23.28
200	200	20%	50%	10%	OFF	7.38	21.56
200	150	20%	50%	10%	OFF	8.77	15.98
200	200	20%	50%	10%	ON	4.83	21.71
200	150	20%	50%	10%	ON	4.99	16.63

Table 1: Table with the results of some parameter tuning of the default genetic algorithm. The template TSP problem with 67 cities is used. The second to 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. #IND = #INDIVIDUALS, #GEN = #GENERATIONS, PR. MUT = MUTATION PROBABILITY, PR. CROS = CROSSOVER PROBABILITY, LP DET = LOOP DETECTION, DIST = OPTIMAL DISTANCE.

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 2.

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 2. 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 2 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 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.

#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**

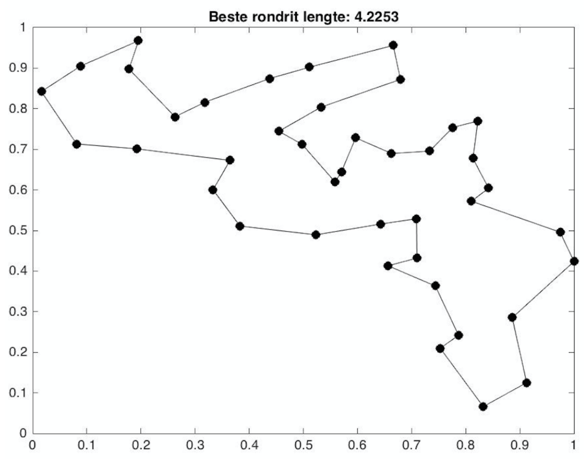
Table 2: Table with the results of some parameter tuning of the genetic algorithm. A path representation is used with order crossover and inversion mutation. For every set of parameters 10 runs are calculated. The template TSP problem with 67 cities is used. The second last column contains the averages 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.

3 Benchmark Problems

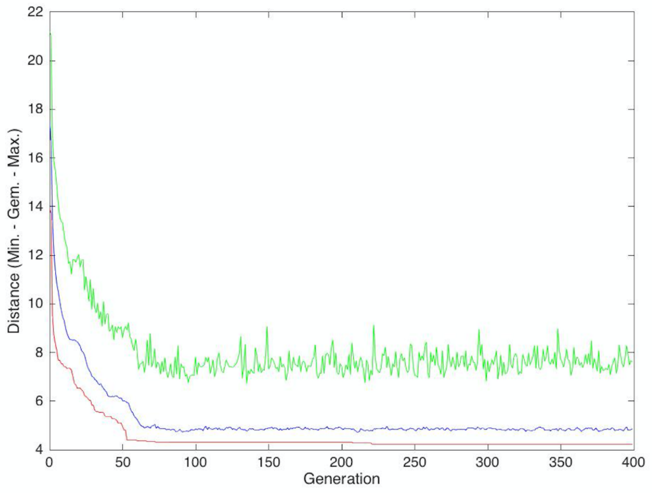
In this section we will test the performance of our designed algorithm (path representation) using some larger benchmark problems. Again we tested the parameter values in table 2. We came to the conclusion that the parameter values found in table 2 are also good parameter values for larger benchmark problems. We executed the tests on the benchmark problem containing 380 cities (`bcl380.tsp`). It is certainly not 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 good 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 a good solution very quickly.

We also did a performance comparison of the original GA (adjacency representation) with the newly designed GA (path representation). For both algorithms their best found set of parameter values are used. In figure 2a and 2b 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 3a and 3b 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 3a 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 3a 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. This could indicate a low variety in the population, which can result in decreased performance.

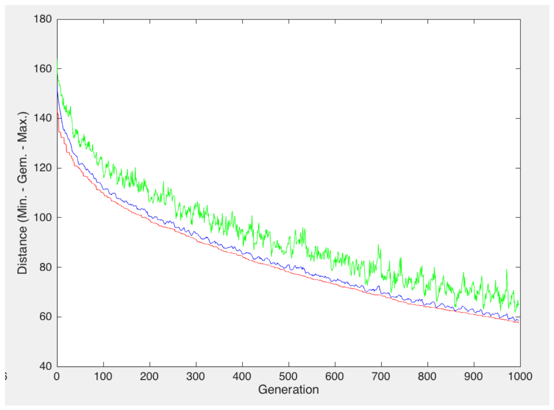


(a) The best tour found for the TSP problem. The distance of the tour is 4.2253.

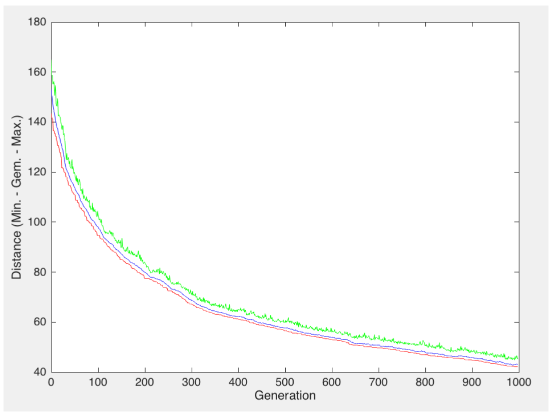


(b) The fitness values (tour distances) for the best(red), average(blue) and worst(green) candidate solution in every generation.

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.

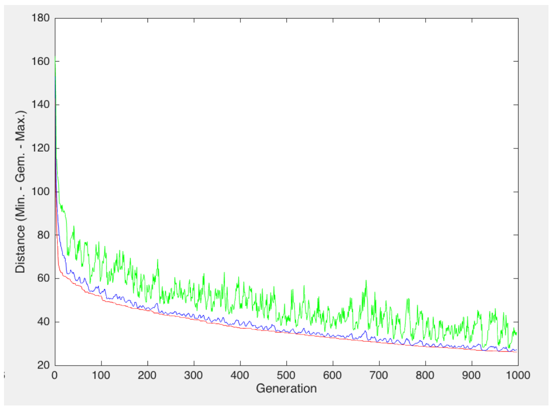


(a) **Adjacency representation (original GA). Best tour distance found: 57.14, computation time: 209 seconds**. PR. MUT = 10%, PR. CROS = 50% , ELITE = 10%. 200 individuals and 1000 generations.

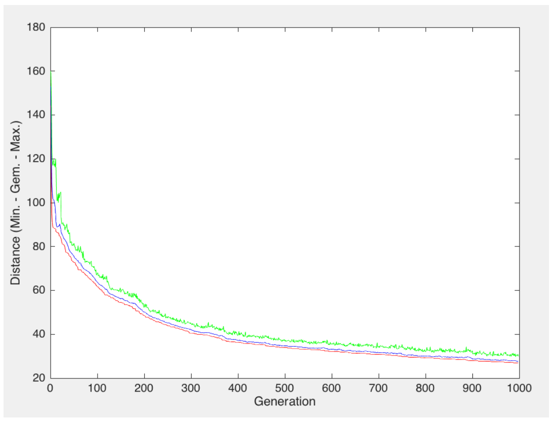


(b) **Path representation (new designed GA). Best tour distance found: 42.1, computation time:357 seconds**.PR. MUT = 8%, PR. CROS = 80% , ELITE = 15%. 200 individuals and 1000 generations

Figure 2: The benchmark TSP problem `bcl380.tsp` with 380 cities is solved here with both the template program and the new representation, loop detection is off for both.



(a) **Adjacency representation (original GA). Best tour distance found: 26.2,computation time:226 seconds**.PR. MUT = 10%, PR. CROS = 50% , ELITE = 10%. 200 individuals and 1000 generations.



(b) **Path representation (new designed GA). Best tour distance found: 26.8,computation time: 387 seconds**. PR. MUT = 8%, PR. CROS = 80% , ELITE = 15%. 200 individuals and 1000 generations are used.

Figure 3: The benchmark TSP problem `bcl380.tsp` with 380 cities is solved here with both the template program and the new representation, loop detection is off for both.

4 Alternative parent selection

This section will discuss two alternative forms of parent selection: tournament selection and fitness proportionate selection.

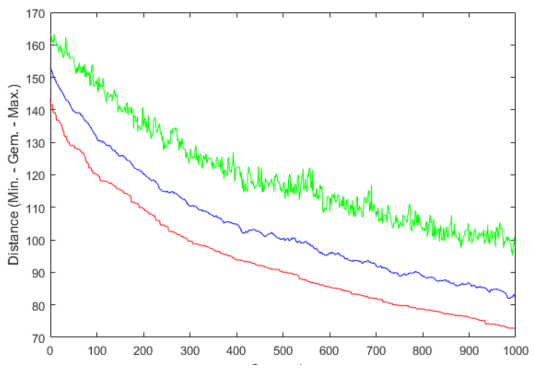
4.1 Fitness proportionate selection

The fitness proportionate selection method assigns to each individual a probability of selection that is directly proportional to its fitness value. This is summed up in formula 1. Because the fitness function is to be minimised, the fitness values are inverted prior to calculating the probabilities.

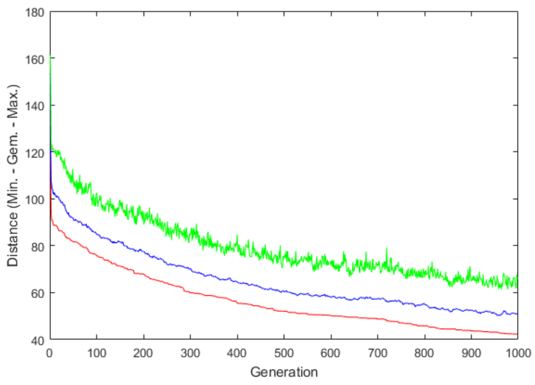
$$P(i) = \frac{\frac{1}{f_i}}{\sum\limits_{j=1}^{\mu} \frac{1}{f_j}} \tag{1}$$

Next, Stochastic Universal Sampling (SUS) is used to select parents based on the probabilities obtained from 1.

After some parameter tuning we performed some of the same benchmarks as in section 3. Figure 4 shows the result of the 380-city benchmark. It is immediately clear that the fitness proportionate selection performs much worse than the previous selection method. A possible explanation would be that the process with fitness proportionate selection tends to stagnate because this selection method tends to suffer from premature convergence. However, looking at the histogram in figure 5 there still seems to be a fairly large amount of variation in the population. It is more likely that the fitness values of the population are too close to each other, resulting in low selection pressure (because similar fitness values lead to similar selection probabilities).



(a) **Loop detection off. Best tour distance found: 72.79, computation time: 390 seconds.**



(b) **Loop detection on. Best tour distance found: 42.36, computation time: 431 seconds.**

Figure 4: The benchmark TSP problem `bcl380.tsp` with 380 cities is solved here using path representation with the following parameters: PR. MUT = 25%, PR. CROS = 60% , ELITE = 5%. 200 individuals and 1000 generations.

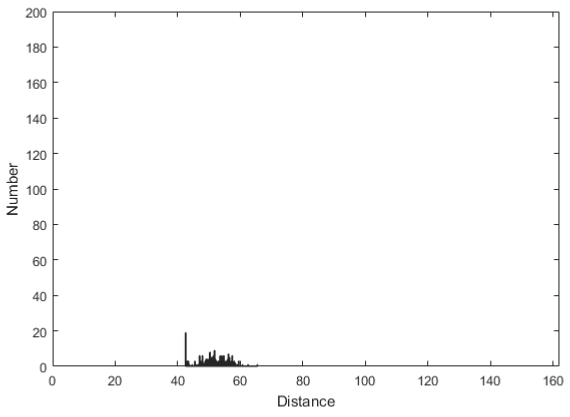
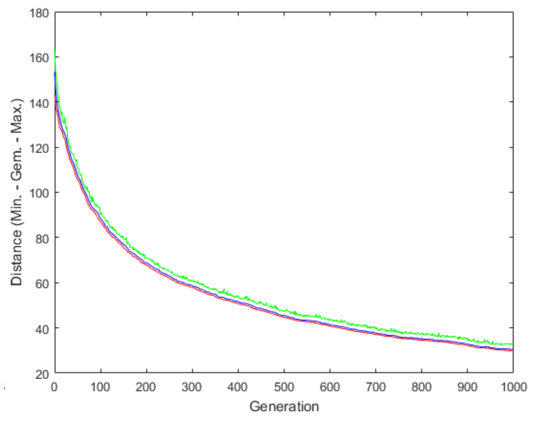
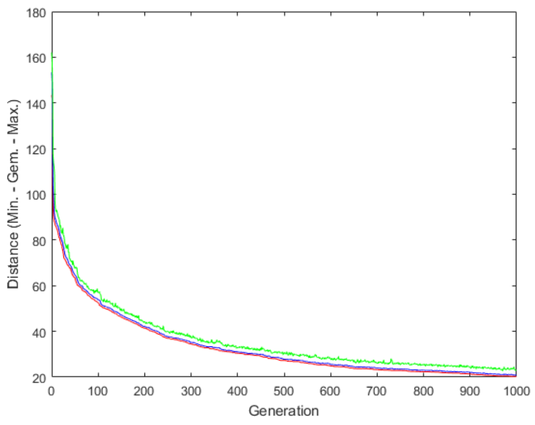


Figure 5: Histogram at 1000 generations using path representations with fitness proportionate selection.



(a) **Loop detection off**. **Best tour distance found: 26.79, computation time: 412 seconds**.



(b) **Loop detection on**. **Best tour distance found: 20.21, computation time: 441 seconds**.

Figure 6: The benchmark TSP problem `bcl380.tsp` with 380 cities is solved here using path representation and tournament selection with the following parameters: PR. MUT = 20%, PR. CROS = 60% , ELITE = 5%, TOURN. SIZE = 5. 200 individuals and 1000 generations.

4.2 Tournament selection

Tournament selection selects parents by randomly choosing groups of n members of the population. From each group the individual with the best fitness value is selected as a parent. This continues until enough parents have been selected. For a full explanation we refer to [1].

To test this selection mechanism the same method as in the previous section is used. From 6 it is clear that tournament selection does offer an improvement over the default rank selection. Worth noting is that tournament selection performs just as well with loop detection disabled as rank selection does with it enabled. A reason for this increased performance might be that tournament selection (with the right tournament size) strikes a good balance between selection pressure and sufficient probability for less fit members to survive, giving the algorithm greater chances of abandoning local optima that are not near the global optimum.

A downside to the tournament selection is that after a large amount of generations, the variety in the population becomes extremely low. Figure 7 shows over half of the population sitting at the same fitness (and thus likely representing the same tour). This reduces the effectiveness of the algorithm and might cause it to get stuck in a local optimum after all. The use of mechanisms to increase population diversity (e.g. supbpopulations) might give further improvements to the algorithm.

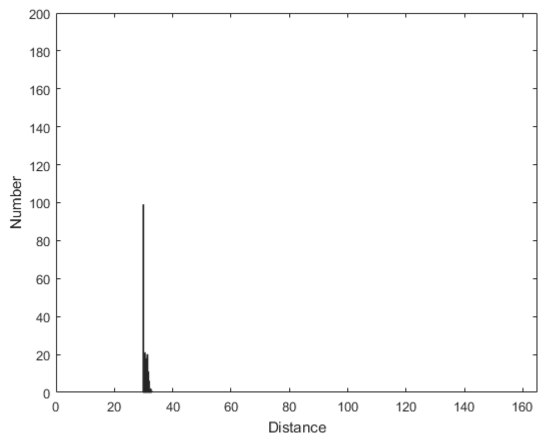


Figure 7: Histogram at 1000 generations using path representations with tournament selection.

5 Conclusion

Unfortunately we have to conclude that we did not manage to make a large improvement to the genetic algorithm template. The use of path representation did only slightly improve fitness but not by much. Tournament selection does seem like a viable selection mechanism, resulting in a moderate improvement to the algorithm's performance.

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.

```matlab
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
```

Implementation of pfitness proportionate selection

This function performs fitness proportionate selection on the given population with given fitness values.

```
function FitnV = FPS(ObjV)

% calculate the Fitness proportional selection (FPS) values
    based on the
% absolute lengts of the tours.

% ObjV is a column vector containing the lengths of the tours.
inverted = 1./ObjV;
sum_lengths = sum(inverted);

% First calculate the ObjV(i)/sum(ObjV)
% Secondly transform the values by f(x) = -x+1 (minimization
    problem)
FitnV = inverted/sum_lengths;
```

Implementation of tournament selection

This function performs tournament selection on the given population with given fitness values.

```
function [ SelCh ] = tournament_selection( Chrom, ObjV, TSIZE, GGAP)
%tournament_selection This function selects a number of parents using the
%tournament method.
%Input Parameters:
%   Chrom - The pool from which mating candidates should be selected
%   ObjV - The fitness values for the Chrom pool
%   TSIZE (optional) - the tournament size. This parameter determines the
%   size for each tournament round, default = 4
%   GGAP (optional) - the genreational gap, default = 1

    DEFAULT_TSIZE = 4
    if (nargin < 2), error('Not enough input parameter'); end

    % Identify the population size (Nind)
   [NindCh,Nvar] = size(Chrom);
   [NindF,VarF] = size(ObjV);
   if NindCh ~= NindF, error('Chrom and FitnV disagree'); end
   if VarF ~= 1, error('FitnV must be a column vector'); end

   if nargin < 3, TSIZE = DEFAULT_TSIZE; end
   if nargin > 2,
       if isempty(TSIZE), TSIZE = DEFAULT_TSIZE;
       elseif isnan(TSIZE), TSIZE = DEFAULT_TSIZE;
       elseif length(TSIZE) ~= 1, error('TSIZE must be a scalar');
       elseif (TSIZE <= 1), error('GGAP must be a scalar bigger than 1');end
    end

   if nargin < 4, GGAP = 1; end
   if nargin > 3,
      if isempty(GGAP), GGAP = 1;
      elseif isnan(GGAP), GGAP = 1;
      elseif length(GGAP) ~= 1, error('GGAP must be a scalar');
      elseif (GGAP < 0), error('GGAP must be a scalar bigger than 0'); end
   end

  % Compute number of new individuals (to select)
```

```
    NSel=max(floor(NindCh*GGAP+.5),2);

     % Select individuals from population
    SelCh = zeros(NSel, Nvar);
    for irun = 1:NSel,
        [sample, idx] = datasample(ObjV, TSIZE, 'Replace', false);
        [M, I] = min(sample);
        SelCh(irun, :) = Chrom(idx(I(1)), :);


    end

end
```