# On Solving Travelling Salesman Problems by Genetic Algorithms

**Heinrich Braun**

Institut für Logik, Komplexität und Deduktionssysteme, Universität Karlsruhe
Postfach 6980, D 7500 Karlsruhe, Deutschland, e-mail: braun@ira.uka.de

## Abstract

We present a genetic algorithm for solving the traveling salesman problem by genetic algorithms to optimality for traveling salesman problems with up to 442 cities. Mühlenbein et al. [MGK 88], [MK 89] have proposed a genetic algorithm for the traveling salesman problem, which generates very good but not optimal solutions for traveling salesman problems with 442 and 531 cities. We have improved this approach by improving all basic components of that genetic algorithm. For our experimental investigations we used the traveling salesman problems TSP (i) with i cities for i = 137, 202, 229, 318, 431, 442, 666 which were solved to optimality in [CP 80], [GH 89].

We could solve medium sized traveling salesman problems with up to 229 cities in < 3 minutes average runtime on a SUN workstation. Furthermore we could solve traveling salesman problems with up to 442 cities optimally in an acceptable time limit (e.g. the average runtime on a SUN workstation for the TSP (431) is about 30 minutes). The greatest examined problem with 666 cities could be approximately solved by constructing a tour with length 0,04% over the optimum.

## 1 Introduction

The application of genetic algorithms for combinatorial optimization problems was already intensively studied in the early seventies ([Re 73], [Sc 77]). Nowadays there is a growing interest in this field in order to find efficient algorithms for parallel computers [Da 85], [Da 87], [Go 89]. In the field of optimization problems the genetic algorithms can coordinate the cooperation of a population of optimum's-searchers [Gr 85], [Gr 87]. Especially for the traveling salesman problem Mühlenbein et al. [MGK 88], [MK 89] have proposed a genetic algorithm, which generates very good but not optimal solutions for traveling salesman problems with 442 and 531 cities.

We have developed this approach further by improving the underlying basic operators. Our basic genetic algorithm works roughly as follows:

```
Initialize population of optimum's searchers
while evolution do
begin
   choose parent1 and parent2;                          {selection}
   offspring := combination (parent1, parent2);         {crossover}
   optimize-local (offspring);                          {mutation}
   if suited(offspring)                                 {survival of the fittest}
   then replace optimum's searcher with worst fitness by offspring
end;
```

In the following sections we present firstly our solution of efficiently implementing the basic operators of this genetic algorithm and secondly our insular genetic algorithm. A more detailed version of our approach can be found in [Br 90].

# 2 Basic Operators

## 2.1 Crossover Operator

Mühlenbein et al. used in their approach order crossover, which were already proposed in [Da 85] and [OSH 87] :

**Order-Crossover (parentstring t,t')**
divide parentstring t in $t_1$ $t_2$ such that length of $t_1$ = k;
$t_2$" := parentstring t' where cities in $t_1$ are removed;
offspring t" := $t_1$ $t_2$" .
{k is constant (e.g. 40% * n), or random (e.g. between 30% * n and 50% * n) }

Using the order crossover the offspring consists of two substrings, one of parent1 and the other of the remaining part of parent2. We generalized this approach by composing the offspring through several substrings destinating alternatively from parent1 and parent2:

**Crossover (parentstring t,t')**
$t_1$ := parentstring t ; $t_2$ := parentstring t' ; t" := ε {ε = empty string};
while $t_1 \neq \varepsilon$ do begin
divide $t_1$ in $t_{11}$ $t_{12}$ such that length of $t_{11}$ = min {length of $t_1$,k};
append $t_{11}$ to t" ;
$t_1$ := $t_1$ where cities in $t_{11}$ are removed;
$t_2$ := $t_2$ where cities in $t_{11}$ are removed;
exchange $t_1$ and $t_2$ ;
end {while};
{ k is constant (e.g. between 20% * n and 40% * n) }

The advantage of our approach is, that a tougher merging of the "genes" of the parents is obtained, which is desirable especially for large problems.

## 2.2 Mutation - Local Optimization

The efficient implementation of the local optimization is crucial for the efficiency of the genetic algorithm. On the one hand nearly the whole cpu-time is used by the local optimization, because the remaining operators as crossover, selection etc. are very fast operators. On the other hand the genetic algorithm needs a good local optimization operator for solving larger problems (cf. the approach in [OSH 87] without local optimization fails even for problems with only 50 cities).

We used the wellknown 2-opt and or-opt heuristic and tuned them especially for the genetic algorithm. Firstly we use the known neighbouring restriction, which means a 2-opt step respectively or-opt step is only allowed, if at least one new edge connects neighbouring cities (the relation "neighbouring" may be defined asymmetricly by: to each city are the k-next cities neighbouring (e.g. k=10)). This is a good restriction heuristic, because it hindered in our experimental investigation no improvement step and decreases the execution time about one order of magnitude. Secondly we take advantage of the similarity of the offspring with the parent by introducing the concept of activ edges, which are the edges of the offspring not not belonging to the parent. We could show that at each 2-opt and or-opt step at least one edge has to be activ. By this restriction the execution time can be decreased about another order of magnitude.

## 2.3 Initialization of the population

For the initialization of the population we have to construct efficiently a set of tours, which are as different and short as possible, in order to achieve a genpool of great diversity and good genes. Therefore one constructs tours with a greedy heuristic and improve this by a tour improvement heuristic. Clearly it is not desireable, to use dedicated heuristics as greedy heuristic (e.g. convex hull) and as improvement heuristic (e.g. Lin Kernighan), because the major improvements will be efficiently done by the genetic algorithm.

As tour improvement heuristic we used the 2-opt and or-opt algorithms (cf. mutation - local optimization). Furthermore we used as an efficient and easy greedy heuristic the well known nearest neighbour algorithm, which constructs in the beginning good subtours. Its low overall performance is only caused by its inability for fitting in the last cities properly, which can be easily improved by 2-opt and or-opt (cf. [LLKS 85]).

## 2.4 Selection and Survival of the Fittest

As in natural surroundings it holds on average: "the better the parents the better the offspring" and "the offspring is similar to the parent". Therefore it is on the one hand desireable to choose the fittest individuals more often, but on the other hand not to often, beause otherwise the diversity of the population decreases. In our implementation we select the best individuals 4 times more often than the worst.

Furthermore we only accept an offspring as a new member of the population, if it differ enough from the other individuals, that means here its fitness differ from all the other individuals at least about amount a. After accepting a new individual we remove one of the worst in order to hold the populationsize constant. In our implementation we remove the worst, because the algorithm is not sensible against this selection.

## 2.5 Genetic Algorithm

Concluding we can notate in the following our genetic algorithm:

**Genetic Algorithm (GA) for the TSP**

```
for i := 1 to #pop do    {#pop = populationsize (e.g. =100)}  {Generation of the population}
begin
    choose new starting town s;
    tour[i] := nearest-neighbour (starting town = s);
    optimize-local (tour[i])
end;

while evolution do
begin
    choose parent1 and parent2;                                  {selection}
    offspring := crossover (parent1, parent2) ;                  {crossover}
    activate-edges (offspring,parent1);
    optimize-local (offspring);                                  {mutation}
    if    ¬ ∃ i ≤ #pop  | length(tour[i]) - length(offspring) | ≤ a   {survival of the fittest}
      ∧   ∃ i ≤ #pop  length(tour[i]) ≥ length(offspring)
    then  replace the longest tour by offspring
end;
```

## 3 Insular Genetic Algorithm

It is easy to see, that our genetic algorithm always converges, because it converges iff the sum of the lengths of all tours in the population converges (which does never increase and is always positiv). Unfortunately it is impossible to predict for such a probabilistic algorithm, whether it is already converged. In practice the convergence will be assumed, when in a certain number of steps no new suited offspring could be produced. This may have two reasons. Firstly the optimum is already generated, that is there are no better tours available. Or secondly the population is degenerated, that is all tours in the population are very similar and could therefore not produce a new suited offspring. In our experimental investigations we found that especially for large problems the population degenerated before the optimal solution was found.

We developed therefore the island model, where several populations each isolated on an island are optimized by the genetic algorithm until they degenerate (e.g. for #N steps). The degeneration is then removed by refreshing the population on each island through individuals of other island (e.g. neighbouring islands) and the evolution can proceed.

**Insular Genetic Algorithm (IGA)**

```
for j := 1 to #island do    {#island = number of islands}
for i := 1 to #pop do    {#pop = populationsize}        {generation of the island-populations}
begin
    choose new starting town s;
    tour[j,i] := nearest-neighbour (starting town = s);
    optimize-local (tour[j,i])
end;

while evolution do
begin
   for j := 1 to #insel do
     for k := 1 to #N do    {#N = number of steps until degeneration}
      begin
          choose parent1 and parent2;                                    {selection}
          offspring := crossover (parent1, parent2) ;                    {crossover}
          activate-edges (offspring,parent1);
          optimize-local (offspring);                                    {mutation}
          if ¬ ∃ i ≤ #pop | length(tour[j,i]) - length(offspring) | ≤ a  {survival of the fittest}
            ∧ ∃ i ≤ #pop  length(tour[j,i]) ≥ length(offspring)
          then replace the longest tour on island j by offspring
     end {for};

   for k := 1 to #m do    {#m = number of merging steps }                {refreshing}
   begin
        choose randomly i,j,i'j' with tour [i,j] new on island i'
                        and tour [i'j'] new on island i ;
        exchange Tour [i,j] and Tour [i'j']
   end {for};

end {while};
```

We can choose the populationsize small (e.g. #pop = 20 - 50) for fast improvements, because the fast degeneration is removed by the refreshing steps.

# 4 Results

As benchmark tests we choose the following traveling salesman problems:

From [CP 80] TSP318 with 318 cities and
from [GH 89] TSP (i) with i cities , i = 137, 202, 229, 431, 442, 666 .

The following figure shows the typical behaviour of the genetic algorithm (here IGA). Our statistics are based on 50 trials respectively.

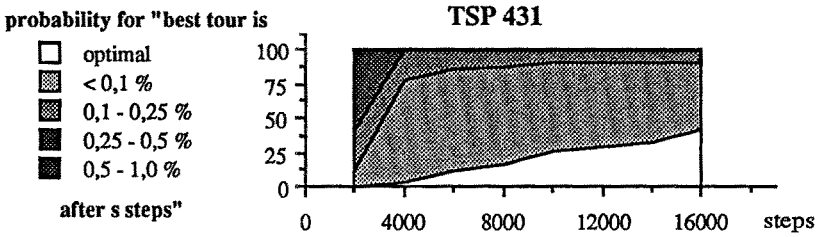


**Figure 1: Statistic of IGA for TSP 431**

The computation time for the generation of one (local optimized) offspring (= 1 step) depends on the problemsize. On a SUN workstation we measured:

| | | | |
|---|---|---|---|
| 22 ms for TSP 137 | 28 ms for TSP 202 | 34 ms for TSP 229 | 38 ms for TSP 318 |
| 52 ms for TSP 431 | 57 ms for TSP 442 | 93 ms for TSP 666 | |

In our experimental investigations we measured as average runtime until receiving the optimal solution: with GA for TSP202 84 s and for TSP229 136s,
with IGA for TSP318 532 s and for TSP431 2080s

# 5 References


[Br 90] **H.Braun,** *Massiv parallele Algorithmen für kombinatorische Optimierungsprobleme und ihre Implementierung auf einem Parallelrechner,* Dissertation an der Universität Karlsruhe, 1990

[CP 80] **H.Crowder, M.Padberg,** *Solving large-scale symmetric travelling salesman problems to optimality,* Management Science, Vol.26, No.5 1980, 495 - 509

[Da 85] **L.Davies (Ed.),** *Applying adaptive algorithms to epistatic domains* Proc. Int. Joint Conf. on Artificial Intelligence, 1985

[Da 87] **L.Davies (Ed.),** *Genetic algorithms and simulated annealing* Pitman London, Morgan Kaufmann Publishers, CA, 1987

[Go 89] **D.E.Goldberg,** *Genetic algorithms in search, optimization and machine learning,* Reading, MA: Addison Wesley, 1989

[Gr 87] **J.J.Grefenstette (Ed.),** *Proceedings of the second int. conf. on genetic algorithms ,* Cambridge, MA: Lawrence Erlbaum , 1987

[GH 89] **M.Grötschel, O.Holland,** *Solution of large-scale symmetric traveling salesman problems,* To appear: Math.Programming, 1989/90

[LLKS 85] **E.L.Lawler, J.K.Lenstra, A.H.G.Rinooy Kan, D.B.Shmoys (ed.)** *The traveling salesman problem,* John Wiley & Sons, 1985

[MGK 88] **H.Mühlenbein, M.Gorges-Schleuter, O.Krämer,** *Evolution algorithms in combinatorial optimization,* Parallel Computing, Vol.7, 1988, 65-85

[MK 89] **H.Mühlenbein, J.Kindermann,** *The dynamics of evolution and learning - towards genetic neural networks,* Connection. in Perspective, Pfeiffer ed., 1989

[OSH 87] **I.M.Oliver, D.J.Smith, J.R.C.Holland** *A study of permutation crossover operators on the travelling salesman problem,* Proc. of the 2. Int. Conf. on Genetic Algorithms, 1987, 224 -230

[Re 73] **I.Rechenberg,** *Evolutionsstrategie - Optimierung technischer Systeme nach Prinzipien der biologischen Information,* Frommann Verlag, Stuttgart, 1973

[Sc 77] **H.P.Schwefel,** *Numerische Optimierung von Computermodellen mittels der Evolutionsstrategie,* Birkhäuser, Basel, 1977