Deutsches
Forschungszentrum
für Künstliche
Intelligenz GmbH

Technical
Memo
TM-97-01

GeneTS: A Relational-Functional Genetic Algorithm for the Traveling Salesman Problem

Markus Perling

August 1997

Deutsches Forschungszentrum für Künstliche Intelligenz
GmbH

Postfach 20 80
67608 Kaiserslautern, FRG
Tel.: + 49 (631) 205-3211
Fax: + 49 (631) 205-3210

Stuhlsatzenhausweg 3
66123 Saarbrücken, FRG
Tel.: + 49 (681) 302-5252
Fax: + 49 (681) 302-5341

This work has been supported by a grant from The Federal Ministry of Education, Science, Research, and Technology (FKZ ITWM-4040).

ISSN 0946-0071

GeneTS: A Relational-Functional Genetic Algorithm for the Traveling Salesman Problem

Markus Perling

August 6, 1997

Abstract

This work demonstrates a use of the relational-functional language RelFun for specifying and implementing genetic algorithms. Informal descriptions of the traveling salesman problem and a solution strategy are given. From these a running RelFun application is developed, whose most important parts are presented. This application achieves good approximations to traveling salesman problems by using a genetic algorithm variant with particularly tailored data representations. The feasibility of implementing sizable applications in RelFun is discussed.

Contents

1 Introduction

Important arguments for using declarative languages are the conciseness of program specifications and support for efficient high-level program development. RelFun is a declarative programming language that integrates the relational and functional paradigms, and we want to demonstrate how RelFun combines the benefits of languages only embodying one of these concepts. Furthermore, we want to show the feasibility of implementing serious projects in RelFun. Both is done by giving an informal algorithmic description of our GeneTS[1] application and systematically developing it into a complete RelFun program.
GeneTS searches good solutions for the "traveling salesman" problem (TSP) using a genetic algorithm (GA). We chose the TSP because it is well-known and constitutes a challenging testbed for various optimization methods, as well as, like in our case, for programming paradigms. Also, the combination of TSP and GA is a well-discussed subject in the literature, see, e.g., [Mic96], [Gol89], on which our discussion is based (also see [RS94]); therefore our program will be comparable to many other implementations. We will not detail here any discussion whether our application is an evolution program or a genetic algorithm (see also [Mic96]): it doesn't use binary coding like GA's in their original definition, but it still uses string coding (or, lists); therefore we will speak of a GA. For direct comparisons between GAs in different declarative languages one will find many (although mostly experimental and therefore undocumented) implementations in Prolog, Lisp, Eiffel, Scheme, and others in the WWW (for a Lisp implementation refer to [Koz93]).
For further reading about the TSP, we suggest a recent text, [JM97]. It discusses several optimization methods applied to the TSP, including genetic algorithms, and can be used as backlink to earlier references. For remarks on the origins of the TSP see chapter 10 in [Mic96] and the footnote at the beginning of [DFJ54]. The latter text is also the earliest reference we found; 'it is shown that a tour across 49 US cities has the shortest road distance'.
This work is divided into two main parts: section 2 informally describes the TSP and GAs, section 3 successively presents the conversion to their RelFun representation. There are two appendices. Appendix A contains a pictorial sample trace of GeneTS applied to a chessboard-like city map, where the optimization process is visualized as an increase of ordering towards an optimal route. In appendix B one finds the complete source of the GeneTS program.

2 The "Traveling Salesman" Problem and Genetic Algorithms

In this section an introduction to the basic concepts of applying genetic algorithms to the TSP is given. Originally, the idea for the project came from [Gol89], and we will follow the language used in this book. Later, our implementation was modified to fit to chapter 10 of [Mic96], which describes the application of evolutionary algorithms to the TSP.

2.1 The "Traveling Salesman" Problem

The general TSP is to plan, as for a salesperson, the shortest possible route between a number of cities. We neglect most of the restrictions that in real situations must be taken into account (e.g. there may not exist a connection from each city to every other), and assume that the following holds:

- each city is connected to every other city,
- each city has to be visited exactly once,
- the salesman's tour starts and ends at the same city

[1]GENEtic Traveling Salesman
from Webster's NewWorld Dictionary: **ge · net** n. [ME. < OFr. *genette* < Sp. *gineta* < Ar. *jarnayt*] **1.** any of a genus (*Genetta*) of small, spottet African animals related to the civet **2.** its fur

Using these assumptions, the heading 'traveling salesman' should be interpreted generically here, encompassing, e.g., also the movement of a robot arm processing a circuit board.
The assumptions lead to a very simple model for the problem: a single tour[2] can be represented by an enumeration of cities; by labeling each city with a number, a valid tour is expressed as a permutation of the numbers $1, \ldots, m$, if we consider m cities. E.g.

$$4, \quad 5, \quad 6, \quad 3, \quad 2, \quad 1$$

describes a tour starting at city number 4, going across cities 5, 6, 3, 2, 1, and ending at city 4. We assume that there exists a map of which the distances between each pair of cities can be read off.
The search space for finding the shortest path across m cities consists of all possible permutations of the numbers $1, \ldots, m$ and has therefore a size of $\frac{1}{2}(m-1)!$; the NP-hardness of the TSP was proven in [GJ79]. In order to still cope with this, we give up trying to find an optimal solution. Thus, our task is to find a reasonably fast algorithm generating satisfying approximations.

2.2 A Bit of Genetics

It is assumed that the reader is already a little familiar (e.g. senior class level) with a few fundamental biological facts concerning evolutionary processes; therefore, the main purpose of this subsection is to make our terms clear.
Writing down number sequences – like the routes above – brings the structure of chromosomes to mind: single 'letters' from a finite alphabet (A,G,C,T in real chromosomes; the numbers $1, \ldots, m$ in the TSP), become stringed together. The elements of the alphabet don't have any meaning in their own, only their ordering within the chromosome has. An organism's properties will be determined by subsequences of these chromosomes, "genes", whose specific contents are called "alleles".
In most higher species, individuals possess two samples of each chromosome, one inherited from each parent; we call this "diploidity", in contrast to a "haploid" set of genomes. In a diploid genome set, each of the duplicate genomes can make contributions to the individuals appearance, but often only one of both is expressed. This depends on the allele of the corresponding gene, and we call this "dominance" of one allele over another, or, in the opposite case, an allele is "recessive" (one remembers here Mendel's laws).
We distinguish between an organisms appearance, the "phenotype", and its genetic information, the "genotype", because the phenotype of an organism is not fully determined by its genotype, but as well is influenced by its environment. Also, recessive genomes belong to the genotype, but not to the phenotype.
A measure for the success of an individual's interaction with its environment, its "fitness", is the number of its descendants: the better an individual copes with the environment, the more descendants it will have. A descendant inherits its properties from its parents, therefore the scions of successful individuals with high probability will also be successful and themselves have more descendants.
One can regard a whole population of individuals as an information pool, each individual's genes representing a set of informations about how to cope with the environment. Genes, and therefore the information, of fitter individuals will spread statistically over the whole population during the succession of generations, and information possessed by less fit individuals will gradually get lost.
If genes within the population wouldn't occasionally be altered, at last the whole population would converge to consist of individuals being exact copies of the initially fittest individual. The alteration of an individuals genetic information is called "mutation". For simplicity, we regard here only two kinds of mutation mechanisms: on the one hand, genetic information of an individual can be altered during its lifetime and, on the other hand, genetic informations of two different individuals can be exchanged during the mating process between the descendants chromosomes. We call the last case "crossover".

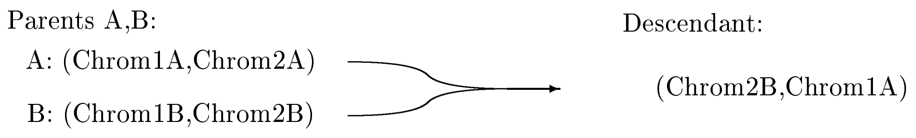
[2]From now on, 'tour' or a 'route' will always abbreviate a salesperson's journey.

2.3 Genetic Algorithms and the Traveling Salesman

Based on our informal depiction above, we describe our approach for solving the TSP. A more formal model is developed, which will form the basis of our implementation.
The purpose of genetic algorithms is to simulate evolutionary processes and to apply them to optimization tasks. Because of the observation that evolution in a changing environment leads successively to well adapted organisms, the assumption is brought up that we can analogously expect well ‘adapted’ solutions. Roughly sketched, we generate a set of random quality solutions, and let an evolution-like process take place that adapts this set to our optimization ‘environment’.
First thing to do when employing a genetic algorithm, is to reformulate our optimization problem so that it can be represented by a fixed-length sequence of parameters $P = \{g_1, \ldots, g_n\}$. Then the optimization function is a function of these parameters: $f_E(x_1, \ldots, x_m) : P \times \cdots \times P \longrightarrow \mathbb{R}$, parametrized by an environment E that we regard to be fixed. If the optimization function is applied to a chromosome, the fitness (see below) of an individual can be obtained. For our TSP, we have the reformulation already done:

- P is the set of m cities to be visited, represented as numbers
- routes over these cities are taken as chromosome
- a chromosome consists of only one gene, whose alleles are any permutation of numbers $1, \ldots, m$; we speak from here analogously of genes, chromosomes, or sequences
- the environment E is a map that contains the distances between each pair of cities
- the optimization function f_E is the route length that has to be minimized.

We regard here a diploid chromosome set, where an allele that represents a shorter tour will dominate and is the only one to be taken into account to determine an individual’s fitness.
A descendant of two individuals is created by selecting randomly one chromosome from each parent and combining them to a new individual:

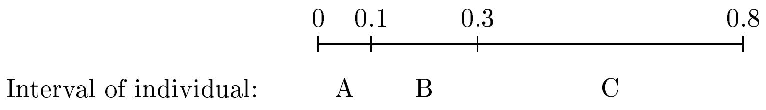


An individuals life cycle is reduced to birth and procreation. So we have a straightforward optimization scheme: we employ a discrete time measure and permit only non-overlapping generations; each time interval represents an optimization step as follows:
Given a population of n members,

- select $n-1$ times pairs of individuals out of the population and create $n-1$ descendants
- calculate the fitness of each descendant
- form a new population from the $n-1$ descendants and include additionally the fittest member of the previous population
- replace the old population with the new
- find the fittest member of the new population and regard it as the actual optimization result

The survival of a generation’s fittest member is called “elitist” model and garantuees a monotone descending sequence of optimization results. The essential of this optimization scheme is the first step, because the fitness is strictly associated with the probability of being selected for mating during one optimization cycle. We remain to a simple heuristics for obtaining an individual’s fitness, the so-called *roulette wheel selection.* If the optimization function f_E is applied to each individual, fitnesses (values for each member of the population) $f_i, i = 1, \ldots, n$, can be obtained. The probability of being selected next time for mating is: $p_i = \frac{f_i}{\sum_{j=1}^{n} f_j} \in [0,1]$, $\sum_i p_i = 1$. In

our TSP, a natural order within the population is given by the route length. We regard an individual representing a shorter tour as fitter than an individual representing a longer tour. Therefore, we get f_i simply by calculating the inverse route length. For a tour length L_i, the fitness is $f_i = \frac{1}{L_i}$ and the corresponding probabilities are $p_i = \frac{1}{L_i}(\sum_{j=1}^{n} \frac{1}{L_j})^{-1} \in [0, 1]$.
To select a parent, the probabilities are associated with disjunct subintervals of length $\frac{1}{L_i}$ of the interval $I = [0, \sum_{j=1}^{n} \frac{1}{L_j}]$. If a random number within I is picked, it lies in a subinterval from which the corresponding individual can be obtained.
For example, the population may consist of three individuals A,B,C, representing tours of length 10, 5, and 2 length units and having fitnesses 0.1, 0.2, 0.5. They are mapped onto the interval $0 \ldots 0.8$:



If, for example, the random value 0.2342 is generated, individual B is selected because the value lies in the corresponding subinterval. One checks easily that the interval assignment correlates directly to the individuals' reproduction probability.
Involved in the reproduction cycle are mutation operators. In adaption to the TSP, each of them must be formulated to avoid the generation of dubletts within the chromosomes. We here describe, by example, five operations in their appropriate adaption to our algorithm (changing chromosome positions are underlined):

1. *Swap two random positions within a chromosome.*

Example:
Given a sequence

$$4, 8, 3, \underline{7}, 10, 2, 5, \underline{1}, 6, 9$$

becomes after mutation:

$$4, 8, 3, \underline{1}, 10, 2, 5, \underline{7}, 6, 9$$

2. *Reversion of a chromosome's subsequence.*

Example:
Given a sequence

$$7, 3, 4, \underline{5}, \underline{1}, \underline{2}, \underline{8}, 9, 6, 10$$

becomes after mutation:

$$7, 3, 4, \underline{8}, \underline{2}, \underline{1}, \underline{5}, 9, 6, 10$$

3. *Remove a random position and insert it at another position.*

Example:
Given a sequence

$$4, 8, 3, \underline{7}, 10, 2, 5, 1, 6, 9$$

becomes after mutation:

$$4, 8, 3, 10, 2, 5, 1, \underline{7}, 6, 9$$

4. *Remove a whole subsequence and insert it at another position.*

Example:
Given a sequence

$$7, 3, 4, \underline{5}, \underline{1}, \underline{2}, \underline{8}, 9, 6, 10$$

becomes after mutation:

$$7, \underline{5}, \underline{1}, \underline{2}, \underline{8}, 3, 4, 9, 6, 10$$

5. *Crossover between two chromosomes.*

During the melting of two chromosomes subsequences are exchanged. Each subsequence must have the same length and is reinserted at the same position in the partner chromosome.
To avoid double entries within a chromosome, we use the PMX (partially matched crossover) algorithm, which works as follows:
Given a newly procreated individual,

1. choose randomly start and end positions
2. make copies of the subsequences within both of the individuals chromosomes, starting and ending at the chosen positions
3. begin with the leftmost entry in the first chromosome's original subsequence and compare it, position by position, with the copy of the second chromosome's subsequence; at each position we have a tuple – the entry within the chromosome and the one within the copy – that determines the values to be exchanged in the first chromosome; the same operation is performed on the other chromosome.

The result is a new pair of chromosomes which have exactly exchanged their subsequences within the chosen positions and are, eventually, modified in another positions. For visualization, we give an example:
Given two individuals consisting of the sequences

First:	$7, 3, 4, 5, 1, 2, 8, 9, 6, 10$
	$10, 9, 8, 7, 6, 5, 4, 3, 2, 1$
Second:	$5, 4, 3, 2, 6, 8, 7, 1, 10, 9$
	$1, 2, 3, 4, 5, 6, 7, 8, 9, 10$

A descendant could be procreated as combination of the first chromosomes of the individuals in the order listed here:

$$7, 3, 4, 5, 1, 2, 8, 9, 6, 10$$
$$5, 4, 3, 2, 6, 8, 7, 1, 10, 9$$

If we decide to perform a crossover, the following happens:

1. select positions and corresponding subsequences:

$$7, 3, 4, \underline{5}, \underline{1}, \underline{2}, 8, 9, 6, 10$$
$$5, 4, 3, \underline{2}, \underline{6}, \underline{8}, 7, 1, 10, 9$$

2. make copies of the subsequences:

$$5, 1, 2$$
$$2, 6, 8$$

3. start pairwise processing of the first chromosome with help of the copy of the second chromosome (double underlined positions are processed in the next step, single underlined positions are already modified in previous steps):

$$7, 3, 4, \underline{\underline{5}}, 1, 2, 8, 9, 6, 10$$
$$\underline{\underline{2}}, 6, 8$$

exchange 2 and 5:

$$\begin{gathered} 7,3,4,\underline{2},\underline{\underline{1}},\underline{5},8,9,6,10 \\ 2,\underline{\underline{6}},8 \end{gathered}$$

exchange 6 and 1:

$$\begin{gathered} 7,3,4,\underline{2},\underline{6},\underline{\underline{5}},8,9,\underline{1},10 \\ 2,6,\underline{\underline{8}} \end{gathered}$$

and, at last, exchange 8 and 5:

$$\begin{gathered} 7,3,4,\underline{2},\underline{6},\underline{8},\underline{5},9,\underline{1},10 \\ 2,6,8 \end{gathered}$$

Analogously the other chromosome is examined, and we get as result:

$$\begin{gathered} 7,3,4,\underline{2},\underline{6},\underline{8},\underline{5},9,\underline{1},10 \\ \underline{8},4,3,\underline{5},\underline{1},\underline{2},7,\underline{6},10,9 \end{gathered}$$

One sees that positions 4 to 6 are exactly swapped and that there must be taken care to avoid dublets, so that follow-up modifications take place at other locations.

3 The Implementation in RelFun

3.1 Preliminaries on RelFun

RelFun is a tight relational-functional integration; it cross-extends Horn relations and call-by-value functions just enough to yield a unified operator concept, as follows: (1) Horn relations are extended to return the truth-value `true`. (2) Functions are extended to allow non-deterministic and non-ground calls; this implies, e.g., that functions can be inverted. RelFun possesses many additional concepts, like higher-order operators and sorts, of which not all will be required here (for further reading see [Bol92]). For the TSP application we restrict ourselves to a mostly pure RelFun style; this allows comparisons with languages supporting different paradigms.
RelFun's syntax is Prolog-like in the sense that programs written in Datalog are also in correct RelFun syntax and have the same behaviour; however, for the full pure Prolog subset of RelFun there are differences, e.g. structures are written with "`[]`"-brackets, the Prolog `is` primitive in RelFun is written as `.=`, and RelFun builtins, such as "`/`" (quotient), are always written in prefix notation. Some readers will be familiar with RelFun's capabilities common with Prolog such as lists and variables.
For further understanding, we note some syntactical remarks.

- First, we have facts, which are the same as in Prolog:

  $$\mathrm{op}(\mathrm{arg}_1, \ldots).$$

- We also have Prolog-like rules:

  $$\mathrm{op}(\mathrm{arg}_1, \ldots) \text{ :- } \mathrm{cnd}_1, \ldots, \mathrm{cnd}_n\,.$$

- Unconditional equations return the value of the `&`-preceded expression:

  $$\mathrm{op}(\mathrm{arg}_1, \ldots) \text{ :\& } \mathrm{exp}\,.$$

- Conditional equations return the `& exp` value if all cnd_i succeed:

  $$\mathrm{op}(\mathrm{arg}_1, \ldots) \text{ :- } \mathrm{cnd}_1, \ldots, \mathrm{cnd}_n \;\&\; \mathrm{exp}\,.$$

The first two kinds of operator definitions, implicitely returning `true`, can be regarded as special cases of the latter two.

- A cut (`!`) can be joined with the neck symbol (`:-` or `:&`):

  op($\text{arg}_1, \ldots$) !- $\text{cnd}_1, \ldots$.
  op($\text{arg}_1, \ldots$) !& exp .

Furthermore, active function calls must be distinguished from passive structures, whose arguments become not evaluated. Syntactically, structures are written like function calls except that they use square brackets instead of parantheses. Structures are unified like Prolog structures, but function calls become first evaluated, then unified.
Analogously, a list can be active or passive. Passive lists simply use square brackets, active lists are generated via the primitive `tup(...)` function.
The user interacts with the RelFun system making queries and requesting, Prolog-like, an arbitrary number of solutions; the demand for further solutions is called `more`-request.
Further concepts will be explained at the places they occur.

3.2 How the Program Works

Given the scheme presented in section 2, we now want to implement it in RelFun. In summary, the program should initialize a starting population and iterate by delivering successively new populations.
In our implementation, the user can specify the following parameters:

1. a list of coordinates for an arbitrary number of cities
2. the size of the population
3. the probability that an individual's genes mutate during its life cycle
4. the probability for a crossover during a mating
5. a probability value that allows scaling between haploid and diploid chromosomes

The first four parameters should be clear; probabilites are always values between 0 and 1, inclusively. The last parameter determines the probability for the selection of the dominant chromosome during mating to become part of a descendant. If this value is 0, a diploid population is simulated, what means, the fitness is determined exclusively by the individuals' dominant chromosomes, but with a probability of 50 % each of an individual's chromosomes may be inherited. If the value is 1, the recessive chromosomes are totally ignored and only the dominant ones become inherited.
Initialization consists simply of generating a set of size n of individuals with random properties. After that, the iteration is as straightforward as described in 2.3.

3.3 The Data Structures: Using Structures and Lists

The structure of a population is summarized by the following BNF-like grammar:

```
population    ::= [individual,individual,...,individual]
individual    ::= indiv[route_length,chrom,route_length,chrom]
chrom         ::= [1,2,3,4,....,n] or permutated entries
route_length  ::= number
```

The semantics is:

- the population is represented by a list of arbitrary size
- a single member of the population is a structure `indiv[...]`
- `indiv[...]` contains an individual's two chromosomes together with the corresponding route lengths; the entries are ordered, s.t. always the shorter route and its length are at the first positions

3.4 The Main Loop: Implementing Interaction via Nondeterminism

The program was devised to visually trace the successive generations of our population; after each reproduction cycle the current optimization results will be printed. The kernel function of our application works therefore tail-recursively and consists of two clauses: the first returns the best member of the population so far, and the second, invoked by backtracking, creates a new generation and calls itself with this new population. The user interacts by giving successive `more`-requests until a satisfactory optimization has been reached.
The source shows this in greater detail:

```
ts(Pop,_,_,_,_) :-
        indiv[BestLength,BestRoute,_,_] .= select_best(Pop,nth(Pop,1))
        & tup(BestLength,BestRoute,/(sum(Pop),len(Pop))).
ts(Pop,Map,Mut_rate,Cross_rate,Better_rate) :-
        New_pop .= next_generation(Pop,Map,Mut_rate,Cross_rate,Better_rate)
        & ts(New_pop,Map,Mut_rate,Cross_rate,Better_rate).
```

The function's first argument is the population, the second the table of distances between the cities, the other arguments will be explained below.
The first clause makes use of the utility functions `select_best` and `sum`[3]; `select_best` scans the whole population exactly once and returns the individual whose first entry contains the shortest route. `ts` returns the result of `select_best` and the population's average route length. In practice, the `ts` function is best called via a testing function, which automatically generates a population of a desired size and the distances table:

```
test(Plan,Pop_size,Mut_rate,Cross_rate,Better_rate) :-
        Map .= generate_distmap(Plan),
        Len .= /(len(Plan),2)
        &  ts(init_pop(Pop_size,Len,Map,init_list(1,Len)),
              Map,Mut_rate,Cross_rate,Better_rate).
```

Its arguments are a list of city coordinates, the demanded size of the population, the mutation rate, the crossover rate, and a weight parameter for scaling between haploid and diploid genomes, which is described in detail below. The `test` function generates the distance table using `generate_distmap` and the coordinates table delivered by `plan`:

```
plan2() :& [
        03.5,00.0,12.5,00.0,00.0,02.0,06.0,02.5,04.0,03.0,
         ...  % some further lines containing coordinates are omitted
        11.5,15.0,03.5,16.5,03.0,18.5,05.5,18.5,10.5,20.5].
```

`init_list(1,Len)` generates a list containing the numbers 1 to `Len`. `init_pop` creates the initial population of the demanded size.

3.5 Individuals: Self-evaluating Data Structures

A useful programming technique in RelFun is giving passive structures and active function calls the same (constructor and function) name. An example in the TSP application is the name `indiv`. If we use it as a constructor for a passive data structure, we cannot rest function calls into its argument positions. However, we can define a function of the same name that will just evaluate to its own call (with its call-by-value arguments recursively evaluated) as a passive structure [4].

```
indiv(L1,R1,L2,R2) :& indiv[L1,R1,L2,R2].
```

Now, one can, e.g., more elegantly write `indiv(+(A,B),X,Y,Z)` instead of `W .= +(A,B), indiv[W,X,Y,Z]` and nevertheless get a data structure.

[3]also, `/` and `len` are builtin-, and user-defined utility functions, respectively: division and length of a list
[4]similarly, `tup(...)` to `tup[...]` or, shortened, `[...]`

3.6 The Roadmap: Converting a Coordinate List into a Distance Table

We use an utility function to convert a list of coordinate tuples into a coordinate table:

```
generate_distmap(Tab) :& generate_distmap1(Tab,Tab).

generate_distmap1([],_) :& [].
generate_distmap1([X,Y|Rest],T) :&
        tup(generate_distmap2(X,Y,T) | generate_distmap1(Rest,T)).

generate_distmap2(X,Y,[]) :& [].
generate_distmap2(X,Y,[X1,Y1|Rest]) :&
        tup(sqrt(+(*(-(X,X1),-(X,X1)),*(-(Y,Y1),-(Y,Y1)))) |
            generate_distmap2(X,Y,Rest)).
```

`generate_distmap` gets as argument a list containing an even number of real numbers and does some calculating for returning a list of equal-length lists (the table) containing the distances between all cities. The i-th entry in the j-th lists contains the distance between cities i and j.

3.7 The Population: Nested Lists and Structures

A valid initial population of given size `N` is generated by `init_pop`:

```
init_pop(0,_,_,_) :& [].
init_pop(N,Len,Map,Ori_template) :-
        Chrom1 .= randomize_list(Ori_template,Len),
        Chrom2 .= randomize_list(Ori_template,Len)
        & tup(make_order(indiv(route_length(Chrom1,Map),Chrom1,
                               route_length(Chrom2,Map),Chrom2))
              | init_pop(1-(N),Len,Map,Ori_template)).
```

Its arguments are the demanded size of the population, the length of an individual, the roadmap and a template list that has to be randomized. In the context of the TSP application, this is usually a list containing the numbers $1, \ldots, N$. First, two chromosomes are generated with `randomize_list`:

```
randomize_list([],_) :& [].
randomize_list(List,N) :-
        NN .= 1+(random(N)),        % select an element from pattern
        Elem .= nth(List,NN),
        NewList .= kill_nth(List,NN) % remove this element from pattern
        & tup(Elem|randomize_list(NewList,1-(N))). % continue with rest
```

Then, a list is returned with the new individual as head and a recursively generated rest. `make_order` returns a dominance-ordered individual.

3.8 The Mechanics of Reproduction: Selection via Probability

Now we describe the central part of the optimization process. As seen in 3.4 this is the function `next_generation`:

```
next_generation(Population,Map,Mut_rate,Cross_rate,Better_rate) :-
        H .= select_best(Population,nth(Population,1)),       % step 1
        Temp_pop .= mutate(Population,Map,Mut_rate),          % step 2
        Probability_range .= probability_range(Temp_pop),     % step 3
        indiv[_,Test,_,_] .= H        % for determing the number of cities
        & tup(H|mate(Temp_pop,Map,len(Temp_pop),              % step 4 + 5
                  Cross_rate,Better_rate,Probability_range,
                  len(Test))).
```

The arguments are the last actual population, the road map and the probability arguments passed through from the ts-function. The function works in five steps (corresponding to the comments in the source):

1. the fittest member of the population is searched
2. a temporary population is generated from the original population by performing mutations
3. from these temporary population the interval for the roulette wheel selection, as described in 2.3, is determined
4. using this interval for selection, a partial new generation is created from the temporary population by performing mating and crossover
5. joining the resulting partial population with the fittest member of the previous generation gives the final population

Mating is performed by the function `mate`:

```
mate(_,_,1,_,_,_,_) :& [].
mate(Population,Map,N,Cross_rate,Better_rate,Probability_range,Length) :-
        P1 .= select_parent(Population,Probability_range),      % step 1
        P2 .= select_parent(Population,Probability_range),
        indiv[_,R1,_,R2] .= mix(P1,P2,Better_rate),             % step 2
        [R1new,R2new] .= cross_over([R1,R2],Cross_rate,Length), % step 3
        LR1 .= route_length(R1new,Map),                         % step 4
        LR2 .= route_length(R2new,Map)
        & tup(indiv[LR1,R1new,LR2,R2new]
              | mate(Population,Map,1-(N),Cross_rate,
                     Better_rate,Probability_range,Length)).
```

`mate` works recursively and procreates a new individual $n-1$ times (the best member of the preceding generation must be saved, and the population shall remain of constant size). The creation of one individual again happens in four steps:

1. select two parents, using the interval generated in `next_generation`,
2. create a new individual by selecting a chromosome out of each parent; this is done by the function `mix` (see below)
3. perform crossover on this individual (see 3.9.2)
4. determine the fitness of the resulting individual

The individual generated in this way is added to the list that will be returned as the new population.
The mixing of chromosomes is done by a `mix` function:

```
mix(indiv[L1,R1,_,_],indiv[L2,R2,_,_],Better_rate) :-
        <(random(1.0),Better_rate)
        & make_order(indiv[L1,R1,L2,R2]).
mix(indiv[L1,P1,_,_],P2,_) :- =(1,random(2)) & make_order(mix1([L1,P1],P2)).
mix(indiv[_,_,L1,P1],P2,_) :& make_order(mix1([L1,P1],P2)).

mix1([L1,P1],indiv[L2,P2,_,_]) :- =(1,random(2)) & indiv[L1,P1,L2,P2].
mix1([L1,P1],indiv[_,_,L2,P2]) :& indiv[L1,P1,L2,P2].
```

The parameters of `mix` are the two individuals to mate and the parameter `Better_rate`, which should be a value between 0 and 1. `Better_rate` is implemented to allow scaling between a haploid and a diploid genome set: in `mix` a random value[5] is generated and dependent on this

[5]If the argument of the builtin `random` is a non-negative real `r`, it returns a random value between 0 and `r`; if the argument is a non-negative integer `n`, a value between 0 and `n` - 1 is generated.

value, either the first (and therefore better) genomes of each individual are put together, or in mix1, the chromosomes are randomly picked. If Better_rate equals 0, a diploid population is simulated, and if Better_rate equals 1, a haploid population is simulated; in this case the mating mechanism degenerates to a selection of possible crossover partners.

3.9 Mutation and Crossover

Both mutation and crossover alter information that is contained within the population, but each of them taking place at different times. We have taken into account several possible ways mutation can happen, but only one possibility of crossover. Therefore, two different ways of implementation were chosen: for mutation, the use of higher-order constructors is demonstrated, whereas crossover is implemented first-order as usual.

3.9.1 Mutation Operators: Parameterization via Higher-Order Constructors

As shown above, the mutation is performed by the function mutate. It is called by next_generation and returns, for intermediate use, a population with mutated members. mutate is recursively defined: it separates the first individual from the rest of the population and performs randomly selected mutations on one of its chromosomes. The result is returned as a list, whose head is the (possibly) mutated version of this individual and whose tail is recursively generated in the same way, until the population is completely processed:

```
mutate([],_,_) :& [].
mutate([First|Rest],Map,Rate) :-
        P .= random(1.0),
        <(P,Rate)
        & tup(mutate1(First,Map) | mutate(Rest,Map,Rate)).
mutate([First|Rest],Map,Rate) :& tup(First|mutate(Rest,Map,Rate)).

mutate1(indiv[_,Better,L,Worse],Map) :-
        =(random(2),1),
        Select .= random(10),
        NewBetter .= mechanism[Select](Better)   % call of mutation operator
        & make_order(indiv(route_length(NewBetter,Map),NewBetter,L,Worse)).
mutate1(indiv[L,Better,_,Worse],Map) :-
        Select .= random(10),
        NewWorse .= mechanism[Select](Worse)     % call of mutation operator
        & make_order(indiv(L,Better,route_length(NewWorse,Map),NewWorse)).
```

First, a random number is generated and tested against the user-given mutation rate. If the test succeeds, the function mutate1 is called with the first member of the population and the roadmap as arguments. mutate1 picks, via random, one of the individual's chromosomes and selects, also by random, a mutation operator mechanism[..]. At this point one notices a different calling scheme: the head functor is no longer an atom but a structure (whose parameters don't necessarily need be instantiated). This allows a syntactical distinction of function parametrization and ordinary function arguments. The random number is selected from the range 0 to 9: there are at most 10 possible mutation operators that are taken into account.
The mechanism function here supports four mutation operators, which are parametrized by the numbers 1 to 4 (which, as shown above, can be generated by the random number generator). An additional clause is added with free constructor parameters whose task is to catch calls to undefined clauses. So the number of supported mutation operators has only to be known at the place where they are defined.
We discuss here two of the mutation operators:

```
mechanism[0](Route) :-              % reversion
        Length .= len(Route),
        Pos1 .= 1+(random(Length)),
        Pos2 .= 1+(random(Length))
        & reverse_sublist(Route,Pos1,Pos2).
mechanism[0](Route) :& Route.

mechanism[1](Route) :- ... % swap two ids, rest of clause omitted here
mechanism[2](Route) :- ... % replace id, rest of clause omitted here

mechanism[3](Route) :-             % replace subsequence
        Length .= len(Route),
        P1 .= +(random(1-(Length)),2), P2 .= +(random(1-(Length)),2),
        Pos1 .= min(P1,P2), Pos2 .= max(P1,P2),
        Head .= get_sublist(Route,1,1-(Pos1)),
        Tail .= get_sublist(Route,Pos2,Length),
        Middle .= get_sublist(Route,Pos1,1-(Pos2)),
        HT .= uni(Head,Tail),
        L .= len(HT), Pos .= +(random(1-(L)),2),
        H .= get_sublist(HT,1,1-(Pos)), T .= get_sublist(HT,Pos,L)
        & uni(H,uni(Middle,T)).
mechanism[3](Route) :& Route.

mechanism[N](Route) :- NN .= mod(N,4) & mechanism[NN](Route).
```

The first is the reversion operator that selects two random positions within a chromosome and reverses the corresponding sublists. The second also selects a sublist, cuts it out, selects a new position within the remaining list, and inserts the cut out sublist at this position [6].

3.9.2 Crossover: PMX

Crossover is performed by the function `cross_over`. It follows exactly the scheme presented in section 2.3.

```
cross_over([P1,P2],Rate,Length) :-
        P .= random(1.0),
        <(P,Rate)
        & cross_over1(P1,P2,1+(random(Length)),1+(random(Length))).
cross_over([P1,P2],_,_) :& [P1,P2].

cross_over1(P1,P2,Pos1,Pos2) :-
        <(Pos2,Pos1)
        & cross_over1(P1,P2,Pos2,Pos1).
cross_over1(P1,P2,Pos1,Pos2) :-
        Sub1 .= get_sublist(P1,Pos1,Pos2),
        Sub2 .= get_sublist(P2,Pos1,Pos2)
        & tup(map_subseq(P2,Sub1,Pos1),map_subseq(P1,Sub2,Pos1)).
```

`cross_over` first generates a random value and checks it against the user-given mutation probability. If the random value is higher, `cross_over` acts as the identity operator. Else, `cross_over1` is invoked with the sequences on which to perform crossover and two random positions as arguments. `cross_over1` selects, via the utility function `get_sublist`, the corresponding subsequences and calls the function `map_subseq`, which does the swapping:

```
map_subseq(L,[],_) :& L.
map_subseq(L,[First|Rest],Pos) :-
        NewPos .= get_pos(L,First)
        & map_subseq(swap_elements(L,Pos,NewPos),Rest,1+(Pos)).
```

[6]`len`: length of list, `uni`: union of lists

4 Conclusions

The source parts explained in the body of this paper constitute approximately 50 percent of our completely implemented GeneTS application, which consists of about 240 lines of program code. The omitted code parts are either the ones explicitely mentioned in the text or ordinary auxiliary routines. Our experience with similar C implementations of genetic algorithms indicate a factor of 4 to 5 in reduction of program size.

Through the development of GeneTS we showed that serious projects can be realised in RelFun (Appendix A), while benefitting considerably from RelFun's high-levelness (Appendix B). Of course, GAs are not a common RelFun application. Because RelFun does not possess any array data type, GA-typical array-like operations have to be simulated with lists. Therefore, the efficiency of GeneTS currently does not compete with implementations in imperative languages like C: functional list manipulation in such cases is less efficient than in-place array updates. This behavior may lead to further optimization considerations towards in-place updates for lists along the lines of [HB85, CH97] in our (WAM-)compiler and emulator combination.

In further work, GeneTS can serve as prototype for a more generalized GA mechanism; RelFun's higher-order capabilities, which we have not used here, can, combined with head-operator parameterization, support the implementation of classes of mutation, crossover and other operators, which can be instantiated by specifying top-level parameters.

References

[BEH+96] Harold Boley, Klaus Elsbernd, Hans-Guenther Hein, Thomas Krause, Markus Perling, q Michael Sintek, and Werner Stein. RFM Manual: Compiling RELFUN into the Relational/Functional Machine. Document D-91-03, DFKI GmbH, July 1996. Third, Revised Edition.

[Bol92] Harold Boley. Extended Logic-plus-Functional Programming. In Lars-Henrik Eriksson, Lars Hallnäs, and Peter Schroeder-Heister, editors, *Proceedings of the 2nd International Workshop on Extensions of Logic Programming, ELP '91, Stockholm 1991*, volume 596 of *LNAI*. Springer, 1992.

[CH97] Chih-Ping Chen and Paul Hudak. Rolling Your Own Mutable MADT — A Connection Between Linear Types and Monads. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66, Paris, January 15 – 17, 1997. ACM Press.

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a Large Scale Traveling Salesman Problem. *Operations Research*, 2:393–410, 1954.

[GJ79] M. Garey and D. Johnson. *Computers and Intractability.* W.H. Freeman, San Francisco, 1979.

[Gol89] David E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison Wesley, 1989.

[HB85] Paul Hudak and Adriene Bloss. The Aggregate Update Problem in Functional Programming Languages. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 300–314, New Orleans, January 1985. "ACM".

[HB87] P. Hudak and A. Bloss. Avoiding Copying in Functional and Logic Programming Languages. In *Conference record of the 14th ACM Symposium on Principles of Programming Languages (POPL)*, pages 300–314, 1987.

[HK70] M. Held and R. M. Karp. The traveling salesman problem and minimum spanning trees. *Operations Research*, 18:1138–1162, 1970.

[JM97] David S. Johnson and Lyle A. McGeoch. The traveling salesman problem: a case study. In E. Aarts and J.K. Lenstra, editors, *Local Search in Combinatorial Optimization*, pages 215–310. John Wiley & Sons, 1997.

[Koz93] John R. Koza. *Genetic programming: On the programming of computers by means of natural selection.* MIT Press, Cambridge, Mass., 1993.

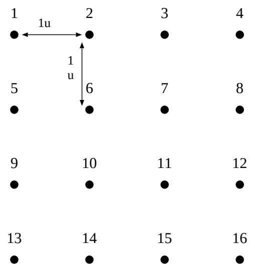
[Mic96] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs.* Springer Verlag, 1996.

[Per97] Markus Perling. RAWAM - A Relfun Adapted WAM. Technical report, Universität Kaiserslautern, perling@dfki.uni-kl.de, 1997.

[RS94] N. J. Radcliffe and P. D. Surry. Fitness Variance of Formae and Performance Prediction. In Whitley, L. D. and Vose, M.D., editor, *Foundations of Genetic Algorithms III.* Morgan Kaufmann Publishers, 1994.

A A Sample Trace of the 16-city Unit TSP

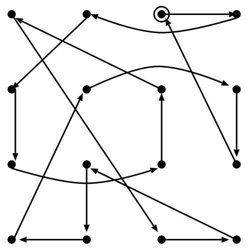
We will give an example trace of the optimizing process by considering a chessboard-like arrangement of 16 cities whose respective distances to their cartesian neighbors are 1 lenght unit each:



It can be easily seen that the shortest route is 16 length units, since the distance between two neighboring cities in all rows and columns is exactly one length unit. Therefore, an optimal tour consists exclusively of horizontal and vertical route segments. The optimization process will become visible by decreasing an original "perturbation" towards a regularly shaped tour. In other words, after each step that generates a better solution than the previous one, there are less tour sections going transversely or overleaping any cities.
First, we begin with a random population:

```
rfc-p> test(plan2(),100,0.4,0.1,1.0)
[26.964037,[3,4,2,5,9,11,7,1,15,16,10,14,13,6,8,12],32.7205]
```

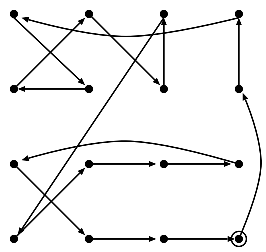
The arguments to the `test` function are the plan corresponding to the set of cities illustrated above, a population size of 100 individuals, mutation and crossover probability of 0.4 and 0.1, respectively, and the value 1.0 indicates a complete haploid genome set. The first result is the length of the shortest path within this population of 26.96 length units, the average route length of the population of 32.72 units, and the shortest route. The generated solution here is the best one out of a completely random set of possible solutions:



The first city of the generated route is city 3, here marked with a circle. One sees that this tour is far from being optimal. Now three optimization steps are invoked:

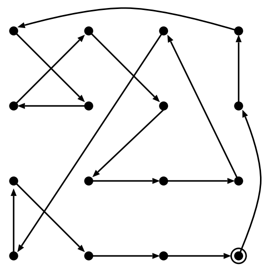
```
rfc-p> more
[26.964037,[3,4,2,5,9,11,7,1,15,16,10,14,13,6,8,12],31.086747]
rfc-p> more
[26.964037,[3,4,2,5,9,11,7,1,15,16,10,14,13,6,8,12],29.923002]
rfc-p> more
[25.676619,[16,8,4,1,6,5,2,7,3,13,10,11,12,9,14,15],28.855871]
```

In each step the average route lenght is decreased; the third step evaluates an individual representing a shorter tour than the fittest one of the first generation:

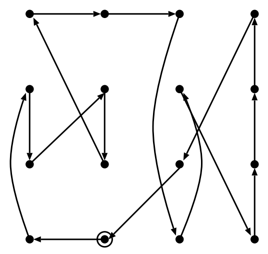


The result is slightly better, but still contains unnecessary indirections. We will leave the next 3 improvements uncommented:

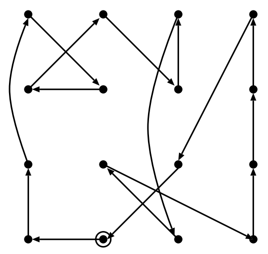
```
rfc-p> more
[25.676619,[16,8,4,1,6,5,2,7,3,13,10,11,12,9,14,15],28.12788]
rfc-p> more
[24.912687,[16,8,4,1,6,5,2,7,10,11,12,3,13,9,14,15],27.429164]
```



```
rfc-p> more
[24.536631,[14,13,5,9,6,10,1,2,3,15,7,16,12,8,4,11],26.810604]
```



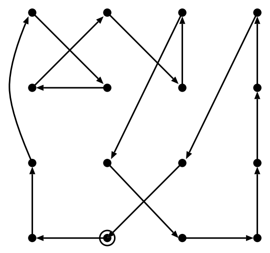
```
rfc-p> more
[23.543204,[14,13,9,1,6,5,2,7,3,15,10,16,12,8,4,11],26.339632]
```



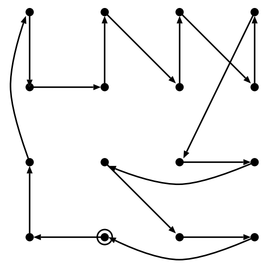
```
rfc-p> more
[23.543204,[14,13,9,1,6,5,2,7,3,15,10,16,12,8,4,11],25.94079]
rfc-p> more
[23.543204,[14,13,9,1,6,5,2,7,3,15,10,16,12,8,4,11],25.502932]
rfc-p> more
[23.543204,[14,13,9,1,6,5,2,7,3,15,10,16,12,8,4,11],25.266099]
```

```
rfc-p> more
[23.543204,[14,13,9,1,6,5,2,7,3,15,10,16,12,8,4,11],24.875117]
rfc-p> more
[21.543204,[14,13,9,1,6,5,2,7,3,10,15,16,12,8,4,11],24.617705]
```

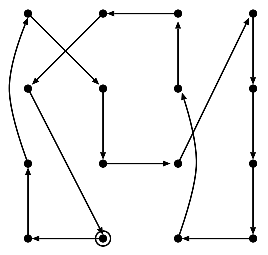
This solution looks, compared to the initial solution, considerably more regular:



```
rfc-p> more
[21.543204,[14,13,9,1,6,5,2,7,3,10,15,16,12,8,4,11],24.309476]
rfc-p> more
[21.543204,[14,13,9,1,6,5,2,7,3,10,15,16,12,8,4,11],24.0268]
rfc-p> more
[21.543204,[14,13,9,1,6,5,2,7,3,10,15,16,12,8,4,11],23.548967]
rfc-p> more
[21.478709,[14,13,9,1,5,6,2,7,3,8,4,11,12,10,15,16],23.256684]
```

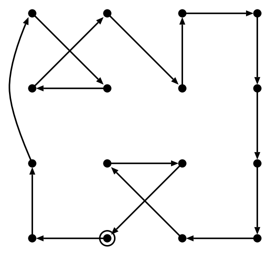


```
rfc-p> more
[21.478709,[14,13,9,1,5,6,2,7,3,8,4,11,12,10,15,16],22.826243]
rfc-p> more
[21.300563,[14,13,9,1,6,10,11,4,8,12,16,15,7,3,2,5],22.522295]
```



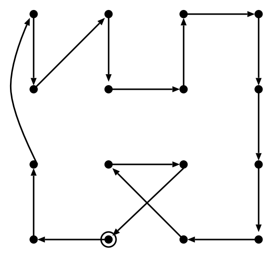
```
rfc-p> more
[21.300563,[14,13,9,1,6,10,11,4,8,12,16,15,7,3,2,5],22.205901]
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],21.754329]
```

Here, we can recognize, what later will be characteristical for the whole population: the route is divided into a lower and an upper half that are only connected by two tour segments between cities 9 and 1, and cities 4 and 8, respectively:



```
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],21.663663]
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],21.270039]
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],21.035227]
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],20.802204]
rfc-p> more
[19.071068,[14,13,9,1,6,5,2,7,3,4,8,12,16,15,10,11],20.454172]
rfc-p> more
[18.242641,[14,13,9,1,5,2,6,7,3,4,8,12,16,15,10,11],19.967564]
```

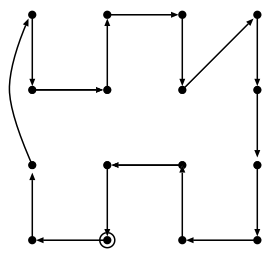
In this result, a tour improvement is achieved by a more optimal course in the upper half:



Compared with the previous result, we observe that the only change in the chromosome was the replacement of city 6 from position 5 to position 7.

```
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],19.456669]
```

Here, the lower half was optimized:

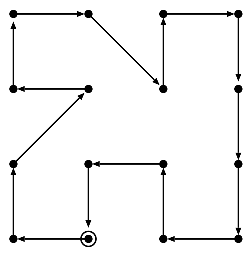


The overleap of city 5 seems to be a common genetic defect to the most individuals in the population. Because we have now a result near to the optimum, it takes some more steps until an individual accumulates enough rearrangements of its genetic information compatible with the achieved optimization results:

```
rfc-p> more
```

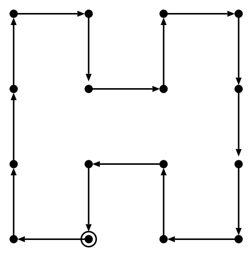
```
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],19.228112]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],19.091763]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],18.848007]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],18.53091]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],18.427471]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],18.04062]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.936131]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.725987]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.521909]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.505341]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.472203]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.453209]
rfc-p> more
[17.414214,[14,13,9,1,5,6,2,3,7,4,8,12,16,15,11,10],17.430782]
rfc-p> more
[16.828427,[14,13,9,6,5,1,2,7,3,4,8,12,16,15,11,10],17.410782]
```

Now the overleap of city 5 has vanished and we have found an almost well-formed tour across the 16 cities:



```
rfc-p> more
[16.828427,[14,13,9,6,5,1,2,7,3,4,8,12,16,15,11,10],17.399066]
rfc-p> more
[16.828427,[14,13,9,6,5,1,2,7,3,4,8,12,16,15,11,10],17.352203]
rfc-p> more
[16.828427,[14,13,9,6,5,1,2,7,3,4,8,12,16,15,11,10],17.302914]
rfc-p> more
[16.0,[14,13,9,5,1,2,6,7,3,4,8,12,16,15,11,10],17.241909]
```

The last result leads to an optimal tour. It differs from the previous one by replacing city 6 from position 4 to position 7:



Our program has found one of the symmetrically equivalent optimal tours, reflecting our choice of a city map that can be connected by a regularly shaped route.

B The Complete GeneTS Source

```
%%% ts.rfp - find a nearly optimal solution of the
%%% traveling salesman problem using a genetic algorithm

%%% the algorithm uses a population of user-defined size;
%%% each individual possesses a diploid genome set, each
%%% representing a tour through all cities;
%%% the phenotype of an individual is represented by the
%%% shorter tour. Each new generation is forme by random
%%% selection and mixing of parents and their genomes

%%% HOW TO USE:
%%% 1. the program needs an ENORMOUS amount of memory,
%%%    allocate as many memory cells as you can
%%% 2. type test(pln,p,q,r), where pl is a map of
%%%    coordinate pairs, n is an integer, p,q,r values
%%%    between 0 and 1;
%%%    n is the population size,
%%%    p the probability for each individual that a
%%%      mutation takes place
%%%    q the probability for a crossover during a mating
%%%    r determines the probability for the selection
%%%      of the better of an individuals chromosomes
%%%      for mating
%%% 3. the length of the shortest route, the route
%%%    itself and the average route length of the
%%%    whole population is printed; results will
%%%    be successively generated by 'more' requests


%%%%%%%% test function

test(Plan,Pop_size,Mut_rate,Cross_rate,Better_rate) :-
        Map .= generate_distmap(Plan),
        Len .= /(len(Plan),2)
        &  ts(init_pop(Pop_size,Len,Map,init_list(1,Len)),
              Map,Mut_rate,Cross_rate,Better_rate).


plan1() :& [
    03.5,00.0,12.5,00.0,00.0,02.0,06.0,02.5,04.0,03.0,
    09.5,04.0,04.5,05.0,02.5,05.5,07.0,05.5,04.0,06.0,
    01.0,06.5,07.5,08.0,12.0,07.0,02.0,07.5,03.5,07.5,
    06.5,07.5,07.0,09.0,05.5,09.5,11.0,10.0,09.9,11.9,
    02.0,11.5,07.5,12.0,10.5,13.0,02.5,13.5,07.0,15.0,
    11.5,15.0,03.5,16.5,03.0,18.5,05.5,18.5,10.5,20.5].

plan2() :& [
    0.0,0.0,1.0,0.0,2.0,0.0,3.0,0.0,
    0.0,1.0,1.0,1.0,2.0,1.0,3.0,1.0,
    0.0,2.0,1.0,2.0,2.0,2.0,3.0,2.0,
    0.0,3.0,1.0,3.0,2.0,3.0,3.0,3.0].


%%%%%%%%  main function

%% generates subsequently new generations and prints
%% the shortest tour, its length, and the average
%% tour length of the population

ts(Pop,_,_,_,_) :-
    indiv[BestLength,BestRoute,_,_]
      .= select_best(Pop,nth(Pop,1))
    & tup(BestLength,BestRoute,/(sum(Pop),len(Pop))).
ts(Pop,Map,Mut_rate,Cross_rate,Better_rate) :-
    New_pop .= next_generation(Pop,Map,Mut_rate,
                               Cross_rate,Better_rate)
    & ts(New_pop,Map,Mut_rate,Cross_rate,Better_rate).

%%%%%%%% activation function for a passive structure

indiv(L1,R1,L2,R2) :& indiv[L1,R1,L2,R2].


%%%%%%%% initialization routines

%% generates from a N-element sequence of
%% coordinate pairs a N*N tableau of distances
%% between each coordinate tuple using
%% euclidean metrics

generate_distmap(Tab) :& generate_distmap1(Tab,Tab).

generate_distmap1([],_) :& [].
generate_distmap1([X,Y|Rest],T) :&
        tup(generate_distmap2(X,Y,T) |
            generate_distmap1(Rest,T)).

generate_distmap2(X,Y,[]) :& [].
generate_distmap2(X,Y,[X1,Y1|Rest]) :&
        tup(sqrt(+(*(-(X,X1),-(X,X1)),*(-(Y,Y1),-(Y,Y1))))
            | generate_distmap2(X,Y,Rest)).


%% initialize a whole population of size Indnum and
%% a genome length of Indsize

%% structure of an individual: indiv[L1,Chr1,L2,Chr2]
%% L1,L2 are the lengths of the tours represented by
%% the chromosomes Chr1 and Chr2. The shorter tour
%% is always at the first place.

init_pop(0,_,_,_) :& [].
init_pop(N,Len,Map,Ori_template) :-
        Chrom1 .= randomize_list(Ori_template,Len),
        Chrom2 .= randomize_list(Ori_template,Len)
        & tup(make_order(
                indiv(route_length(Chrom1,Map),Chrom1,
                      route_length(Chrom2,Map),Chrom2))
             | init_pop(1-(N),Len,Map,Ori_template)).


%% randomizing a list

randomize_list([],_) :& [].
randomize_list(List,N) :-
        NN .= 1+(random(N)),
        Elem .= nth(List,NN),
        NewList .= kill_nth(List,NN)
        & tup(Elem|randomize_list(NewList,1-(N))).


%%%%%%%% generator of successive populations

%% saves the fittest member of the older generation
%% ("elitist variant") and generates N - 1 new members.
%% This process involves all mechanisms like mutation
%% and crossover.

next_generation(Population,Map,Mut_rate,
                Cross_rate,Better_rate) :-
        H .= select_best(Population,nth(Population,1)),
        Temp_pop .= mutate(Population,Map,Mut_rate),
        Probability_range .= probability_range(Temp_pop),
        indiv[_,Test,_,_] .= H
        & tup(H|mate(Temp_pop,
                        Map,
                        len(Temp_pop),
                        Cross_rate,
                        Better_rate,
                        Probability_range,
                        len(Test))).


%% selects for each individual of a population randomly
%% a mutation operator and applicates it to the
%% genome, respectively to the route. the resulting,
%% possibly permutated, routes are merged to a new
%% population

mutate([],_,_) :& [].
mutate([First|Rest],Map,Rate) :-
        P .= random(1.0),
        <(P,Rate)
        & tup(mutate1(First,Map) |
              mutate(Rest,Map,Rate)).
mutate([First|Rest],Map,Rate) :&
        tup(First|mutate(Rest,Map,Rate)).

mutate1(indiv[_,Better,L,Worse],Map) :-
        =(random(2),1),
        Select .= random(10),
        NewBetter .= mechanism[Select](Better)
        & make_order(indiv(route_length(NewBetter,Map),
                           NewBetter,L,Worse)).

mutate1(indiv[L,Better,_,Worse],Map) :-
    Select .= random(10),
    NewWorse .= mechanism[Select](Worse)
    & make_order(indiv(L,Better,
                 route_length(NewWorse,Map),NewWorse)).


%% performs the reproduction cycle of the population.
%% The new population is generated by successive
%% selection of pairs of individuals and exchanging
%% the genome sequences. As an additional mutation
%% operator the crossover mechanism is invoked.

mate(_,_,1,_,_,_,_) :& [].
mate(Population,Map,N,Cross_rate,Better_rate,
     Probability_range,Length) :-
      P1 .= select_parent(Population,Probability_range),
      P2 .= select_parent(Population,Probability_range),
      indiv[_,R1,_,R2] .= mix(P1,P2,Better_rate),
      [R1new,R2new]
          .= cross_over([R1,R2],Cross_rate,Length),
      LR1 .= route_length(R1new,Map),
      LR2 .= route_length(R2new,Map)
      & tup(indiv[LR1,R1new,LR2,R2new] |
            mate(Population,Map,1-(N),Cross_rate,
                 Better_rate,Probability_range,Length)).


%% selects subsequences out of two routes at the
%% same position and exchanges them

cross_over([P1,P2],Rate,Length) :-
        P .= random(1.0),
        <(P,Rate)
        & cross_over1(P1,P2,1+(random(Length)),
                            1+(random(Length))).
cross_over([P1,P2],_,_) :& [P1,P2].

cross_over1(P1,P2,Pos1,Pos2) :-
        <(Pos2,Pos1)
        & cross_over1(P1,P2,Pos2,Pos1).
```

```
cross_over1(P1,P2,Pos1,Pos2) :-
        Sub1 .= get_sublist(P1,Pos1,Pos2),
        Sub2 .= get_sublist(P2,Pos1,Pos2)
        & tup(map_subseq(P2,Sub1,Pos1),
              map_subseq(P1,Sub2,Pos1)).


%% returns the sum of all probability weights;
%% to select a certain individual out of the population
%% each is assigned to an interval. the size of the
%% intervall is 100 times the reciprocal of the length
%% of the phenotypical route. all intervals can
%% be thought subsequential ordered on the real axis
%% beginning at 0. the larger the interval the more
%% likely the random number generator will
%% generate a value within the interval so that
%% the individual will be selected.
%% annotation: possibly the reciprocal is not the
%% most perfect weighting, there may be weights
%% that prefer the fitter individuals, but this would
%% be unnecessarily complicated

probability_range([]) :& 0.
probability_range([indiv[L,_,_,_]|Rest]) :&
        +(probability_range(Rest),/(100,L)).


%% selects an individual out of the population using
%% the interval returned by probability_range
%% (see there)

select_parent(Pop,Range) :- P .= random(Range)
        & select_parent1(Pop,P,0).

select_parent1([I],_,_) :& I.
select_parent1([indiv[L1,R1,X,Y]|_],P,Offset) :-
        <=(P,+(Offset,/(100,L1))) & indiv[L1,R1,X,Y].
select_parent1([indiv[L,_,_,_]|Rest],P,Offset) :&
        select_parent1(Rest,P,+(Offset,/(100,L))).


%% mixes randomly the genome pair of two individuals
%% means: two lists, each with two elements, are mixed

mix(indiv[L1,R1,_,_],indiv[L2,R2,_,_],Better_rate) :-
        <=(random(1.0),Better_rate)
        & make_order(indiv[L1,R1,L2,R2]).
mix(indiv[L1,P1,_,_],P2,_) :- =(1,random(2))
        & make_order(mix1([L1,P1],P2)).
mix(indiv[_,_,L1,P1],P2,_) :&
        make_order(mix1([L1,P1],P2)).

mix1([L1,P1],indiv[L2,P2,_,_]) :- =(1,random(2))
        & indiv[L1,P1,L2,P2].
mix1([L1,P1],indiv[_,_,L2,P2]) :& indiv[L1,P1,L2,P2].


%%%%%%%% common utility functions

%% mutation operators
%% operator 0: reversion of a whole sequence
%% operator 1: swapping of two ids
%% operator 2: placing an id at an another position
%% operator 3: placing a whole subsequence at
%%     an another position
%% operator N: catching of operator numbers bigger
%%     than 2 and mapping them to the first two

mechanism[0](Route) :-
        Length .= len(Route),
        Pos1 .= 1+(random(Length)),
        Pos2 .= 1+(random(Length))
        & reverse_sublist(Route,Pos1,Pos2).
mechanism[0](Route) :& Route.

mechanism[1](Route) :-
        Length .= len(Route),
        Pos1 .= 1+(random(Length)),
        Pos2 .= 1+(random(Length))
        & swap_elements(Route,Pos1,Pos2).
mechanism[1](Route) :& Route.

mechanism[2](Route) :-
        Length .= len(Route),
        Pos1 .= 1+(random(Length)),
        Pos2 .= 1+(random(Length))
        & insertion(Route,Pos1,Pos2).
mechanism[2](Route) :& Route.

mechanism[3](Route) :-
        Length .= len(Route),
        P1 .= +(random(1-(Length)),2),
        P2 .= +(random(1-(Length)),2),
        Pos1 .= min(P1,P2),
        Pos2 .= max(P1,P2),
        Head .= get_sublist(Route,1,1-(Pos1)),
        Tail .= get_sublist(Route,Pos2,Length),
        Middle .= get_sublist(Route,Pos1,1-(Pos2)),
        HT .= uni(Head,Tail),
        L .= len(HT),
        Pos .= +(random(1-(L)),2),
        H .= get_sublist(HT,1,1-(Pos)),
        T .= get_sublist(HT,Pos,L)
        & uni(H,uni(Middle,T)).
mechanism[3](Route) :& Route.

mechanism[N](Route) :- NN .= mod(N,4)
        & mechanism[NN](Route).


%% changes the order within a genome in the way
%% that the shorter route is the first element;
%% the shorter route represents also the phenotype
%% therefore only one of the genomes has to be
%% tested to determine the fitness of an indivdual

make_order(indiv[L1,R1,L2,R2]) :-
        <(L2,L1)
        & indiv[L2,R2,L1,R1].
make_order(X) :& X.


%% computes the length of a route from a given
%% table which is a 2-dimensual array of
%% real-numbers whose entries are the distances
%% from one city to another

route_length([First|Rest],Map) :&
        route_length1(First,[First|Rest],Map).

route_length1(F,[A],Map) :& nth(nth(Map,A),F).
route_length1(F,[A,B|Rest],Map) :&
        +(nth(nth(Map,A),B),route_length1(F,[B|Rest],Map)).


%% selects the individual which represents
%% the shortest route out of the whole population

select_best([],Best) :- ! & Best.
select_best([indiv[L1,R1,L2,R2]|Rest],
            indiv[L3,R3,L4,R4]) :-
        <(L1,L3)
        ! & select_best(Rest,indiv[L1,R1,L2,R2]).
select_best([_|Rest],Best) :- ! & select_best(Rest,Best).


%% returns the sum of all route lengths in a population

sum([]) :& 0.
sum([indiv[L,_,_,_]|Rest]) :& +(L,sum(Rest)).


%% embeds a sublist in a list and eliminates duplicates

map_subseq(L,[],_) :& L.
map_subseq(L,[First|Rest],Pos) :-
        NewPos .= get_pos(L,First)
        & map_subseq(swap_elements(L,Pos,NewPos),
                     Rest,1+(Pos)).


%%%%%%%% general utility functions

%% reversion of a sublist from element no. Pos1 to Pos2

reverse_sublist(List,Pos1,Pos2) :- >(Pos1,Pos2)
        & reverse_sublist1(List,Pos2,Pos1).
reverse_sublist(List,Pos1,Pos2) :&
        reverse_sublist1(List,Pos1,Pos2).

reverse_sublist1(List,1,1) :& List.
reverse_sublist1(List,1,Pos2) :&
        reverse_sublist2(List,[],Pos2).
reverse_sublist1([First|Rest],Pos1,Pos2) :&
   tup(First|reverse_sublist1(Rest,1-(Pos1),1-(Pos2))).

reverse_sublist2(R,List,0) :& uni(List,R).
reverse_sublist2([First|Rest],L,N) :&
        reverse_sublist2(Rest,[First|L],1-(N)).

%% returns the union of two lists

uni([],L2) :& L2.
uni([First|Rest],L2) :& tup(First|uni(Rest,L2)).


%% returns nth element of a list

nth([First|_],1) :& First.
nth([_|Rest],N) :& nth(Rest,1-(N)).


%% returns length of a list

len([]) :& 0.
len([_|Rest]) :& +(len(Rest),1).


%% initializes list of length Begin - End
%% with integer values Begin ... End

init_list(End,End) :& [End].
init_list(Begin,End) :&
        tup(Begin | init_list(1+(Begin),End)).


%% replaces element of a list at position Pos
%% with Elem

replace_elem([_|Rest],1,Elem) :& [Elem|Rest].
replace_elem([First|Rest],Pos,Elem) :&
        tup(First|replace_elem(Rest,1-(Pos),Elem)).
```

```
%% swaps two elements of a list
%% at positions Pos1 and Pos2

swap_elements(List,Pos1,Pos2) :-
        A .= nth(List,Pos1),
        B .= nth(List,Pos2),
        L .= replace_elem(List,Pos1,B)
        & replace_elem(L,Pos2,A).

%% removes element at position Pos1 and
%% inserts it at position Pos2

insertion(List,Pos1,Pos2) :-
        >(Pos2,Pos1)
        & insertion1(List,Pos1,Pos2).

insertion(List,Pos1,Pos2) :& insertion2(List,Pos1,Pos2).

insertion1(List,Pos1,Pos2) :-
        Elem .= nth(List,Pos1),
        NewList .= kill_nth(List,Pos1)
        & insert(NewList,Elem,1-(Pos2)).

insertion2(List,Pos1,Pos2) :-
        Elem .= nth(List,Pos1),
        NewList .= kill_nth(List,Pos1)
        & insert(NewList,Elem,Pos2).

%% returns list without nth element

kill_nth([_|Rest],1) :& Rest.
kill_nth([First|Rest],N) :&
        tup(First|kill_nth(Rest,1-(N))).


%% returns subsequence of a list from
%% position Pos1 to pos. Pos2

get_sublist(L,1,Pos2) :& get_sublist1(L,Pos2).
get_sublist([First|Rest],Pos1,Pos2) :&
        get_sublist(Rest,1-(Pos1),1-(Pos2)).

get_sublist1([First|_],1) :& [First].
get_sublist1([First|Rest],Pos2) :&
        tup(First | get_sublist1(Rest,1-(Pos2))).


%% returns position of an element in a list

get_pos([First|_],First) :& 1.
get_pos([_|Rest],E) :& 1+(get_pos(Rest,E)).


%% inserts element into list at nth position

insert([First|Rest],Elem,1) :& [Elem,First|Rest].
insert([First|Rest],Elem,N) :&
        tup(First|insert(Rest,Elem,1-(N))).
```

TM-97-0
Technical Mem

GeneTS: A Relational-Functional Genetic Algorithm for the Traveling Salesman Problem

Markus Perling