Adaptive Strategies and the Design of Evolutionary Applications

José Neves **Miguel Rocha** **Hugo Rodrigues** **Miguel Biscaia** **José Alves**

Departamento de Informática
Universidade do Minho
PORTUGAL

Abstract

Over the last few decades, one has observed a remarkable increase, both in the number, and in the quality of approaches to problem solving, with an inspiration on natural evolution. It was shown how the evolutionary processes can be applied to solve scientific and engineering problems, using what is well understood as genetic, or more generally, *Evolutionary Algorithms (EAs)*. However, in the software engineering counterpart, some correlations have not been fully matched, and often a new problem to solve implies the development of an application from scratch. So, how does this apply to problem solving ? By developing a system that will take advantage of the major features of the object-oriented paradigm, using the ANSI/ISO C++ language. By considering several abstraction's levels, therefore encapsulating the most significant building blocks of the *EA*, although leaving the sufficient room for specific implementations, so common in the field. A system that will be reusable, in that, under mild conditions, a wide variety of *EAs* can be tested (e.g., *Evolution Strategies*, *Evolution Programming* and *Genetic Algorithms*), either in their sheer versions, or by considering some processes of hybridization.

1 Introduction

In problem solving, the generality dimension suggests two ways for moving through the problem space - one can make the existing hypothesis more general, or one can make it more specific. These options also suggest two basic schemes for searching of the space of concept descriptions. One may start with the most general hypothesis and, as new instances are found, more specific descriptions are brought forth. One the other hand, one may begin with a very specific hypothesis, moving to more general descriptions, as new facts are observed. Both approaches take advantage of the partial ordering on hypothesis to constrain search, and most machine learning approaches to problem solving have adopted one or the other of these methods. Whereas most problem solving systems have to arrange search according to the procedure referred to above, an exception is the evolutionary approach, which does not use the partial ordering [Hol75], [Hol86], [Fog66] and [Rec73]. This knowledge is acquired on the fly, and used to guide the evolution process; e.g. to reject actions, namely *crossovers* or *mutations*, which are either recorded or suspected to be unsuccessful. It is claimed that on trivial (static) problems, avoiding to repeat errors is significantly less misleading than attempting to repeat successes.

In recent years, one has been observing a remarkable growth in the volume of applications, aiming to tackle an increasing number of problems, in a broader set of domains, such as *Numerical* and *Combinatorial Optimization*, *Design*, *Computer Vision*, *Machine Learning*, *Telecommunications*, *Scheduling and Timetabling*, just to name a few. Furthermore, it is clear a significant trend to combine the capabilities of the *Evolutionary* (or *Darwinian*) paradigm to problem solving, with other techniques and methods, bringing to life the so called *hybrid* systems, already with some good results on their shoulders.

An analysis to the *Evolutionary Computation (EC)* field reveals an enormous diversity in terms of the methodologies, architectures, algorithms and operators used, that obviously leads to alike, but not identical, software, and presents a major obstacle to its reusability. However, a careful look at the wide picture, provided by this kaleidoscope of applications, can provide an useful insight. In fact, it is undeniable that the great majority of the proposed approaches share some common features; i.e., the building blocks that make the core of the *EC* artifacts define a common background, that can be used in terms of software development and analysis.

Indeed, the purpose is straightforward - to identify these building blocks, and to implement them, when building *EC* applications. In order to accomplish this aim, one resorts to the major features of the object oriented paradigm, namely viewing problem solving as a process that goes through an hierarchy of abstraction spaces - each criticality level being a different abstraction space. The solution at each level forms a plan for the solution at the immediate lower level, making the way for the modular and incremental construction of *EC's* applications; i.e., the user only needs to implement the features that are specific to his/her problem, and to take advantage of all the previously stated abstraction spaces.

The system allows for several kinds of use, depending on the task the user has at hand, and the programming skills he/she possesses. Thus, the user can, on the one hand, use the system's knowledge, data structures and operators, providing only the coding of the solutions and the evaluation function, or he/she can, on the other hand, redefine both the data or knowledge's structure and operators, or even the *Evolutionary Algorithm (EA)* itself. A framework is created under which a vast number of paradigms may be studied, including *Evolution Strategies (ES)*, *Evolution Programming (EP)* and obviously *Genetic Algorithms (GAs)*. A special attention is devoted to make sure the system is capable of allowing for the development of *hybrid* systems, in their various forms. So, it is set that one may combine the hierarchy of abstraction spaces that make a problem, it's data structures and operators, with the hierarchy of abstraction spaces that make the *EA*. The language chosen to implement the system was the ANSI/ISO C++ [Str86].

In the next section, it is briefly presented the architecture of the proposed system, and addressed some implementation issues, being given an overview of the process of building an *EC* application. One finishes interposing some questions and pinpoint some directions for future work.

2 The Archetype

The system's architecture is basically made upon three conceptual levels, encapsulating the properties and methods of the main entities presented in it, namely the *individuals*, the *populations*, and the *EAs* (Figure 1). The procedures *decodification* and *evaluation* make the interface between the environment (or the problem state) and the system.

The main question that arises is concerned with the creation of abstraction spaces at these levels, and how to connect the evaluation module with these structures, in a convenient manner. These problems are addressed by using abstract classes, with template fields, at the different abstraction spaces, at the different conceptual levels. *A posteriori*, the templates are assigned

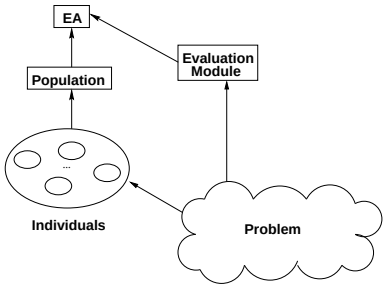

Figure 1: The system's conceptual levels.

with the respective types, where a type assignment to a finite set Γ of type variables $x_i (i = 1, ..., n)$, is a finite list $E = (x_1, T_1), \ldots, (x_n, T_n)$, such that any T_i is legal in Γ.

At the *individual's* conceptual level, one defines an hierarchy of classes, whose root is an abstract class, *Indiv*, with a template field that contains its genotype; i.e., its genetic information. In this way, one sets the doings for any kind of representational scheme, simply by assignment the template with the necessary type, either simple or compound (Figure 2). The classes so far developed contemplate binary genes, order-based representations (with integer genes), and real-valued representations (genes of C++ type float). Other kinds of representational schemes and operators can easily be introduced into the system, when required. An *individual* is given as an object, with a set of shared attributes (e.g., fitness value, size) and a vector of genes of a given type (the chromosome).

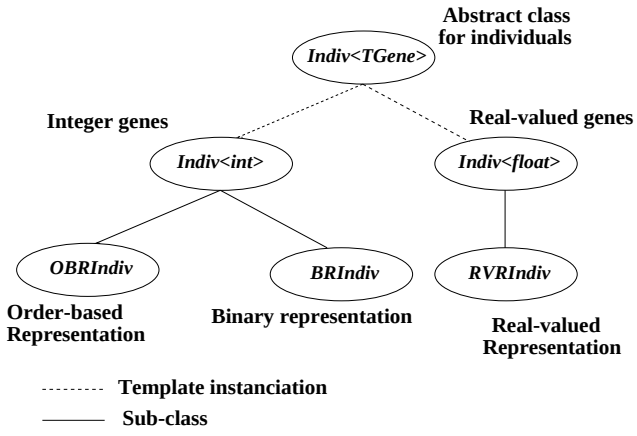

Figure 2: Scheme of the classes developed at the *individual's* level.

A similar strategy is used at the *population's* level, where the basic class is *Popul*, and the template field

keeps the type of the individuals used by the *EA*. A *population* is, therefore, an object containing several attributes (e.g., size, statistics, selection parameters, operator's table), and an array of individuals of a given type, defined as referred above; i.e., as classes at the *individual's* level.

At the upper level one has the *EA*, being possible to define an hierarchy, rooted by a class that takes the *population* as a template field. Its possible instantiations are determined by the set of the classes defined at the *population's* level. An *EA* object includes an attribute of a separate class, *EAPar*, to keep some run-time user defined data, such as the selection procedure and control parameters, the population size, the chromosome size, the information on the operator table, and a link to the evaluation module. At this level one defines the structure of the evolutionary algorithm; i.e., the computations that may occur in each iteration of the *EA*, and the termination criteria. A method, *run()*, is created, setting the default structure of the *EA* (Figure 3); this can be changed by redefining the method into an user defined subclass.

```
BEGIN
  Initialize time (t = 0).
  Generate the individuals that will make
    the initial population (P_0).
  Evaluate these individuals.
  WHILE NOT (end criteria) DO
    Select from the present population (P_t)
      the individuals for reproduction.
    Apply the genetic operators to the parents,
      in order to breed the offspring.
    Proceed on the evaluation of the new individuals.
    Select the offspring to insert
      into the next population (P_{t+1}).
    Select the survivors from P_t
      to be inserted into P_{t+1}.
    Increase current time (t = t + 1).
  END WHILE
END
```

Figure 3: Default structure of the EA.

The last of the functionalities to be incorporated into the system was the *evaluation module*. This makes the interface between the *EA* programming environment, and the user's related applications, clustering into a class (and into sub-classes), the problem specific features of the *EA*. It is mandatory to have a sub-class to class *Eval*, defining the process of decodification; i.e., how to depart from the genotype and arrive at the phenotype (the solution to the target problem). It is also necessary to define the way this solution should be evaluated; i.e., how to assign a fitness value to the individual.

In general, the *EA's* are got by default, thus allowing a less prepared user to take advantage of the system, but also enabling a more experienced one to (re)design several features from scratch and create its own *EAs*.

3 Implementation issues

3.1 Genetic operators

When faced with existing implementations of *EAs*, one is confronted with a multitude of genetic operators, namely the *crossover* and *mutation* ones, intimately related to the representational schemes used at the level of the individuals' genotype.

However, in a vast number of approaches, *crossover* and *mutation* are not the sole operators used. For example, it is common to consider specific optimization operators, as in the so called *Lamarckian* evolution [Whi95]. Other alternatives contemplate the use of inversion operators, social disaster techniques [Kur96], or multi-parent crossover (or orgies) [Muh93], among many others.

In order to cope with all this diversity, and to gain greater flexibility, a scheme was designed to consider, for each application, a table of operators, to be stochastically used at the reproduction phase. To each operator on the table, is associated a range from which it may be selected, via a *roulette wheel* scheme. The sum of these values, for the chosen operators, should be 1 (one):

$$\Gamma = \{(\eta_i, P(\eta_i)) \mid \sum_{i=1}^{n} P(\eta_i) = 1\} \qquad (1)$$

where Γ is the stochastic operator table, η_i is the operator i, $P(\eta_i)$ is the probability of use of operator i, and n is the number of operators. Each operator is also defined in terms of the number of individuals taken as input, the number of individuals it returns, and a value that measures its sense of application; i.e., a genetic operator may be defined as the production:

$$Operator: \; Indiv_1 \times \ldots \times Indiv_m \times Parameters \mapsto Indiv_1 \times \ldots \times Indiv_n$$

where the '$\times$' stands for the *Cartesian product* and '$\mapsto$' means '$\mapsto$'. The described scheme allows one to represent both the *mutation* and *crossover* operators, and also to contemplate the definition of others. Furthermore, it enhances the algorithm by opening the possibility of considering several different operators in the same run, feature that is believed to improve the efficiency of the genetic algorithm in preventing its premature convergence to local optima.

The possibility of self-adaptation and control of the *EA*, by changing the value of some parameters on the fly is also contemplated, although the algorithms used

to do so in a profitable way are still a subject under study [Ang95].

The definition and implementation of the genetic operatos is, obviously, done across the abstraction space, that make the *individual's* conceptual level. General purpose operators, that can be applied to diverse representacional schemes are defined at higher levels, while the more specific ones can be found at the base of the hierarchy. Therefore, specific operators should be defined in the leaf classes.

In run time, whenever an operator is selected to be applied to a given set of individuals (the ancestors) of a given type, the more specific suitable operator is applied.

3.2 Selection

A set of selection procedures are provided, being at the user's shoulders to make the best choice, although the system may act by default. Each of these procedures make an object of the *EA*, being characterized by:

- The problem's type (i.e., it may be a minimization or maximization one);
- The number of the individuals of the population being replaced, per generation;
- The number of ancestors selected for reproduction, per generation;
- The number of offspring created, per generation;
- The selection method used (e.g., roulette wheel, universal stochastic sampling, tournament);
- The normalization process of the fitness values used under the evaluation phase; i.e., before selection (e.g., raw, linear scaling, ranking)x ;
- The process used to convert a problem from a minimization setting, to a maximization one, when necessary;
- The elitism value; i.e., the number of the individuals that are automatically selected to be part of the next generation.

.

3.3 Evaluation function

The integration of the evaluation function, in an *EA*, is quite straightforward. A sub-class to the virtual class *Eval* is created, and an object of this class is included as one of the *EA's* parameters, at the level of the *EA*. This subclass should implement a method, *evaluate()*, that assigns a fitness value to each individual in the population. Conceptually, it should perform a decodification of the chromosome into a solution to the problem, and then evaluate it (Figure 4).

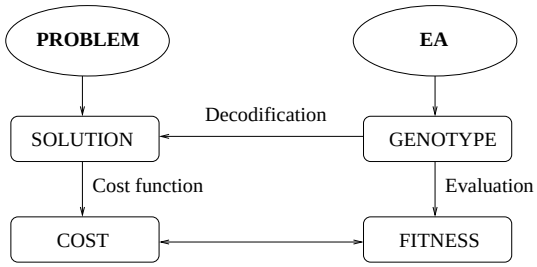

Figure 4: Conceptual scheme of the evaluation procedure.

In terms of implementation, one includes in this class a pointer to a structure containing the necessary data from the problem instance.

3.4 Exception handling

To deal with abnormal situations a constraint handling mechanism has been devised, working like a firewall. The abnormal situations raise exceptions. These are defined in terms of the class *Invariant*, given as:

```
template class <class I, class E>
Invariant(I inv, E* excep)
```

At the implementation level, integer values are used for the types I and E. The exceptions thrown when an abnormal event is detected are captured by an object, and handled by its set of instructions. One handles both general purpose exceptions (e.g., memory allocation errors), and specific ones raised by the user's misuse of the classes provided.

3.5 I/O facilities

The I/O subsystem is based on C++ streams, being device independent (Figure 5). To each class, with input/output operations, two operators were developed:

$$istream\& \ operator >> (istream\& \ is, object\& \ obj) \quad (2)$$

$$ostream\& \ operator << (ostream\& \ os, object\& \ obj) \quad (3)$$

where operator (2) implements the reading operation; i.e., it reads an object (e.g. *Popul* or *Indiv*) from a device, handled by the input stream *is*. In a similar way, through the operator (3), an object is written to a device.

3.6 C++ operators

The features of the C++ language were largely exploited, namely its aptitude for operator handling, here

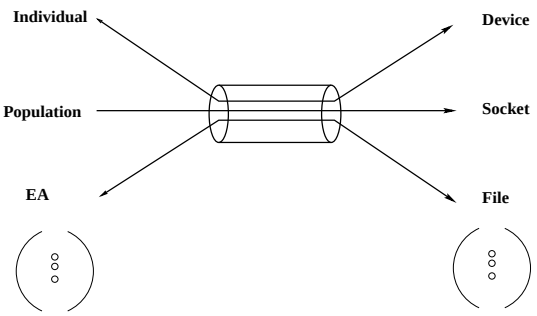

Figure 5: Stream I/O facilities.

materialized through the operators:

- *operator*[] – it pinpoint objects in a container (e.g., a gene in an individual or an individual in a population).
- *operator*+ – it adds an object to a container.
- *operator*– – it removes an object from a container.

On the other hand, operators like *assignment* or *comparison* were generalized to handle different kinds of data types. This made the code to be more legible, and keen for maintenance.

3.7 Termination criteria

The termination criteria is based on the number of generations of the *EA*. Nevertheless, other methods are being studied, in particular the ones based on statistical measures of convergence of the *EA*, and the genetic diversity of the populations.

4 Building evolutionary applications

The framework so far studied may now be used in order to build *EA* applications. One starts presenting the user with a set of alternatives in the use of *EAs* for problem solving, and the necessary steps to be considered at each stage. Finally, a practical example is shown, namely the way one can build a set of different *EAs* for solving the well known *Traveling Salesman Problem (TSP)*.

One may consider, basically, two different approaches to problem solving, according to the basics of *EC*. The first one leads to the development of *EAs*, independently of the problem to solve, creating robust and general-purpose optimization tools. In this case, the framework should provide the necessary representational schemes and operators, at the *individual's* level, and also provide the general structure of the *population* and the *EA*, at the higher levels. For example, choosing a binary representation to the individuals, together with a proper selection of parameters and operators, it should be sufficient to build a simple *GA*, as defined by Holland [Hol75].

To cope with such demands, there are already available some representational schemes, and their associated operators. If the problem fits into one of these schemes, then the task of the programmer, in terms of software development and analysis, is greatly reduced. There is, however, an operation that has to be considered, namely the creation of the evaluation procedure, following the methodology referred to above (Section 3.3).

The results obtained so far with general purpose evolutionary problem solving tools are not yet satisfactory - the solutions to the problem may not be coded with the representational schemes, as the ones provided (Figure 2), the general purpose operators may not be efficient enough, or even the default structure of the *EA* is not the most suitable one. Therefore, some more specific applications, contemplating special representacional shemes and/or genetic operators are needed, as well as hybridization processes, that may contribute to combine the merits of the *EAs*, with those of other techniques. It is a requirement of the system to be flexible enough to give the proper support to each of these scenarios.

The (re)definition of new functionalities in the system is done by a well known technique in object-oriented programming, that involves the creation of sub-classes at the appropriate conceptual levels (*individuals*,*populations*, *EA*, *evaluation*). For example, the inclusion of a new genetic representational scheme or the definition of problem specific operators imply the inclusion of a sub-class at the *individuals* conceptual level, as well as the definition of a non-random initial population implies the inclusion of a new class at the *population's* conceptual layer, or the definition of a non-stationary fitness landscape would require the inclusion of a sub-class into the *evaluation module*.

4.1 Developing EA's to tackle the TSP

The *TSP* is a classic, well known NP-hard problem in *Combinatorial Optimization* - given a set of n cities, seen as a graph, and the costs associated with the travel between each pair, the objective is to find a round-trip of minimal total cost (or length), visiting each node exactly once.

The problem is stated as a n-dimensional cost matrix of values d_{ij}, where the purpose of the exercise is to obtain a n-permutation, such that the sum of the values d_{ij}, for any i and j being i the precedent of j in the sequence, is minimal. It can be defined in terms of *Integer Linear Programming* as follows [Lap91]:

$$Minimize: \quad \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}x_{ij} \quad (4)$$

$$Subject\ to: \quad \sum_{j=1}^{n} x_{ij} = 1, \forall i \quad (5)$$

$$\sum_{i=1}^{n} x_{ij} = 1, \forall j \quad (6)$$

$$x_{ij} \in \{0,1\}, \forall i,j \quad (7)$$

$$\sum_{i,j \in S} x_{ij} < |S|, \forall S \subset V, S \neq \emptyset \quad (8)$$

where (4) defines the objective function in terms of the admissible costs (edges d_{ij}), and a decision procedure given by equations (5), (6), (7), and (8). If x_{ij} is 1 (one), the edge connecting i and j belongs to the solution, otherwise (x_{ij} is 0 (zero)) the contemplated edge is not in the solution. The equations (5) and (6), state that only one edge enters and leaves a given node. In (8), V stands for the set of nodes and $|S|$ for the cardinality of S, a subset of V. Equation (8) restricts the solutions' set, by eliminating those that have cycles, whose length is smaller than n (being n the number of nodes on the graph).

For the purpose of the *EA*, each individual (or chromosome), codes a *TSP* valid tour. The genotype of the individual is built on a sequence of n integers, with no repeated values. The phenotype interpretation of this is fairly obvious, once the position of any allele on the chromosome determines the order by which the node that it codes is visited. An edge is assumed to connect the nodes that are given by the chromosome's last and first values.

This kind of genetic representation is named *Order-Based Representation (OBR)*, being, as referred to above, included in one's library.

As far as a *TSP* instance is considered, the evaluation function will assign, to each individual in the population, a fitness value, a measure of the cost of the solution coded by its genotype.

The first *EA* here developed, to tackle the *TSP*, follows a general purpose strategy. It uses all the given values by default, works with *individuals* of the class *OBRIndiv*, makes no changes to the definitions in the *population's* conceptual level, and uses the default structure for the *EA* (Figure 3). Under these settings it was only necessary to implement a convenient evaluation class, assuming the existence of a *TSP* class (or set of classes), defining the data structures and methods for the problem; i.e., it was created an *EvalTSP* class, sub-class to *Eval*, with a pointer to an object of the *TSP* class, defining the instance of the problem that one intends to solve. In this class it were defined two methods: an evaluation method and a class constructor.

If the case is to combine the *TSP's* solving techniques (or heuristics), with the *EA*, some new functionalities may have to be implemented. If the intention is to develop new genetic operators, which may use knowledge that stems from the *TSP* instance being solved (e.g., intelligent crossover or local optimization operators), there is the need to create a class *TSPIndiv*, subclass to *OBRIndiv*, where this new set of operators can be inserted. One may even consider the possibility of use of heuristics, appropriate to handle the *TSP*, in order to generate its initial solutions ; i.e., the individuals that will make the initial population of the *EA*. This implies the redefinition of the correspondent method, at the level of class *TSPPopul* (a subclass to *Popul*), and also of the process to create each individual, in class *TSPIndiv*.

Other possibility is the application of the *Baldwin effect* [Ack91] to the *TSP*, in which case it is needed a local optimization operator to improve the fitness of the individuals, at each iteration, but, unlike the *Lamarckian* approach, the new solution is not coded back onto the genotype. In terms of implementation, as far as the presented system is concerned, one has only to create, into class *EvalTSP*, a new evaluation procedure; i.e., a new instance to method *evaluate()*.

5 Conclusions and further work

The motivation behind this work stemmed from one's goal to create a development tool, a tool that will make the software so far written to specific evolutionary applications to be reusable in future tasks; i.e., preventing the programmer from beginning his/her work always from scratch. Indeed, the *EA's* development process shall be incremental and modular, allowing for applications of increased conceptual complexity, yet not imposing a similar overhead on the developer's shoulders. It also makes a toolkit that allows a less experienced user to build an *EA*, with a minimum programming effort. The use of the object-oriented paradigm, and a proper identification of the *EA* building blocks, was crucial to the success of one's objectives. Although this work is still undergoing, it is believed that the methodological principles presented make the proper core to the *EC's* software development and analysis processes of the future.

In terms of new developments, it is being considered the creation of spatial structured populations, thus enabling parallel models for the *EAs*. Another area of research is concerned with the on the fly algorithms' self-adaptation, and the definition of proper termination criteria. Indeed, a major problem in *EAs* is the premature convergence to local optima, which is strongly connected with the genetic diversity of the populations. Some techniques to face this problem are being developed and integrated into the system [Roc99]. The integration of *Neural Networks* and *EAs* is another area under consideration, where one's purpose is to tackle more complex problem's domains with non-stationary fitness landscapes (e.g. *Machine Learning* tasks). The issue of diploidy and dominance rules

is yet another issue under study, as well as the so called *Baldwin effect.*

It is also ones aim to test the system and improve its robustness, by developing several *EA's* applications to selected fields. In the *Combinatorial Optimization* field, and apart from the above referred experiments with the *TSP*, one is working in *Graph Coloring, Knapsacking*, *Scheduling* and *Timetabling* problems. In *Numerical Optimization* the aim is to build a general-purpose optimization tool for multi-variable functions, subject to a number of constraints, based on real-gene valued representational schemes.

In conclusion, one can say that the *EC* field is at a turning point. The definition of methodological principles, the sedimentation of processes of software development and analysis, make one's contribution on such an ongoing process.

6 Acknowledgments

Special thanks are due to Eduardo Marques and Orlando Pereira.

References

[Ack91] D.H.Ackley and M.Littman, Interactions between learning and evolution. In C.G.Langton, *Proc. of the 2nd Conf. on Artificial Life*, Addison Wesley, 1991.

[Ang95] P.J.Angeline, Adaptive and Self-adaptive evolutionary computation. In Palaniswami M., Attikiouzel Y., Marks R., Fogel D., Fukuda T. (eds). *Computacional Intelligence: A Dynamic System Perspective*, pages 152-161. IEEE Press, 1995.

[Fog66] L.J.Fogel, A.J.Owens e M.J.Walsh, *Artificial Intelligence Through Simulated Evolution*, John Wiley, New York, 1966.

[Hol75] J.H.Holland, Adaptation in Natural and Artificial Systems, University of Michigan Press, Ann Arbor, 1975.

[Hol86] J.H.Holland, Escaping Brittleness: The Possibilities of General Purpose Learning Algorithms Applied to Parallel Rule-based Systems. In R.S.Michalski, J.G.Carbonell and T.M.Mitchell(eds), *Machine Learning: An Artificial Intelligence Approach*, Vol.II, Morgan Kaufmann, Los Altos, CA, pp.593-624, 1986.

[Kur96] V.Kureichick, A.N.Melikhov, V.V.Miaghick, O.V.Savelev and A.P.Topchy, Some New Features in the Genetic Solution of the Traveling Salesman Problem. In Ian Parmee and M.J.Denham eds. *Adaptative Computing in Engineering Design and Control 96(ACEDC'96), 2nd International Conference of the Integration of Genetic Algorithms and Neural Network Computing and Related Adaptative Computing with Current Engineering Practice*, Plymouth, UK, March 1996.

[Lap91] Gilbert Laporte, The Traveling Salesman Problem: An Overview of Exact and Approximate Algorithms, *European Journal of Operational Research*, 59:231-247, 1992.

[Mic96] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs.* 3rd edition, Springer-Verlag 1996.

[Muh93] H.Mühlenbein and D.Schlierkamp-Vosen, Predictive Models for the Breeder Genetic Algorithm In *Evolutionary Computation*, Vol.I,No.1,pp.25-49, 1993.

[Rec73] Ingo Rechenberg, *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*, Holzboog Verlag, Stuttgart, 1973.

[Roc99] M.Rocha and J.Neves, Preventing Premature Convergence to Local Optima by Random Offspring Generation. Research Report, Departamento de Informatica, Universidade do Minho, Janeiro 1999.

[Str86] Bjarne Stroustrup, *The C++ Programming Language*, 2nd ed., Addison-Wesley, 1986.

[Whi95] Darrell Whitley, Modeling Hybrid Genetic Algorithms. In O.Winter, J.Periaux, M.Galan, P.Cuesta eds. *Genetic Algorithms in Engineering and Computer Science*, pp.203-216, John Wiley, 1995.