Object Oriented Design and Implementation of a General Evolutionary Algorithm

Róbert Ványi

Department of Theoretical Computer Science
Friedrich-Alexander University, Erlangen-Nürnberg
Martensstr. 3, D-91058 Erlangen, Germany
vanyi@cs.fau.de

Abstract. There are many types of evolutionary algorithms, just like genetic algorithms, evolution strategies or genetic programming. Thus there are a lot of implementations of different EAs as well. One can find many tools, programming libraries written in a wide variety of programming languages, implementing several features of Evolutionary Algorithms. However, many of them misses one or other feature by restricting the algorithm to a certain type. The aim of the work presented in this paper is to introduce a general, object oriented EA model, and to present an implementation of it. One of the most important advantages of this system is that it uses a generalized view of EAs and evolutionary operators, thus any combination of different types of evolutionary operators can be used with lots of parameters. The system is also easy to extend, and still easy to understand and use. If one wants to use the already implemented data types and operators, implementing an EA is an easy ride. For example an evolutionary algorithm solving a polynomial approximation problem for real-valued polynomials can be implemented with only 100 lines of C++ code.

1 Introduction

When someone decides to use evolutionary algorithms to solve a problem, a decision is also made, what kind of EA is used. This usually determines the possibly used operators, and the representations. The most common types are

- evolution strategies [Rec73] using real vectors, and operators on real numbers (adding a random number, averaging, and so on),
- genetic algorithms [Hol75] [Gol89] with bitvector representations and bit operators, and
- genetic programming [Koz92] with tree data structures, and tree operators.

However, one may want to combine several properties of these methods and thus many home-brew systems cannot be considered as pure GAs, ESs or GPs. The available programming environments, however, usually consider one of these models, therefore they are not general enough.

In this paper a general EA model is constructed and an object oriented implementation in the programming language ANSI C++ is presented. Advantages of this object oriented C++ implementation are

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 1275–1286, 2004.
© Springer-Verlag Berlin Heidelberg 2004

- *easy to understand*: a small number of clearly defined classes,
- *easy to use*: simple problems can be solved with 100 lines of code,
- *easy to extend*: new representation types and operators can be implemented using inheritance
- *general*: a general view of EAs and operators is used
- *fast*: combines OO and the efficiency of C

1.1 Related Work

There are many implementations of evolutionary algorithms. Many of them was examined, but no general EA implementation was found. SGA-C [SGE94] and GENESIS [Gre84] were implemented in C, thus the advantages of the object oriented programming cannot be used. SGA-C was not meant to be a general tool, and GENESIS concentrates only on GAs as well. Evolvica [Evo] and its predecessor eaLib [eaL] are implemented in Java, and though sometimes Java programs may be fast, the author still prefers C++ over Java. Furthermore eaLib have many classes, making its usage difficult, though there are many examples in the documentation. Evolvica intends to be a programming framework, so it is not only a library. From the libraries implemented in C++ three were examined, GAlib [Wal96], EO [Mer] and GEA [Tót00]. GAlib has many nice features, can handle different individuals and operators, and these can also be extended. But it still does not use a general approach either for the algorithm or for the operators. EO is also a fully-featured implementation, based on C++ templates available for many platforms. However, it also lacks a general EA implementation and the general approach to evolutionary operators, though different types of algorithms can be implemented by subclassing a given algorithm class. GEA was developed to overcome some disadvantages of these systems, but it does not consider giving a general model for EAs or operators either.

As it can be seen, none of these libraries, systems use a general EA view, and all of them use more or less the usual approach for operators, that is having a crossover (recombination) followed by a mutation. To create a system having these missing features is the aim of the work presented here. On the other hand these systems have many nice features, not considered by the general EA design and implementation introduced in this paper, but the detailed comparison of these systems is beyond the scope of this paper.

We proceed as follows. First a general evolutionary algorithm model is introduced. In Section 3 an object oriented design is given for the general EA model. In the pseudocode an object oriented notation is used, that is attributes and methods of objects are written as $objectname.attr$, and $objectname.func()$. This design is discussed further with concrete implementation details in Section 4. In Section 5 some examples are given how a specific problem can be solved with the implemented general EA. Finally in Section 6 conclusions are drawn and some future plans are mentioned.

2 General Evolutionary Algorithm

To design a general evolutionary algorithm a very simple approach is used. The EAs store a set of objects, do something with them, and create a new set of objects. Doing something with the objects means that several evolutionary operators and selections are applied on them. Sometimes selections are handled as operators, but in our case they are distinguished from each other. When searching in literature several types of selections can be found. One can select individuals to be parents, one can select individuals from the generated individuals, that is from the offsprings to be survivors. One can also directly copy individuals into the offsprings or even into the survivors. The latter one is called elitism. Sometimes there are not enough survivors to make up a new population, so new individuals may be created randomly. The selections use a fitness function to measure the goodness. To create a general evolutionary algorithm, it is allowed for each selection to have an own fitness function.

Using all these components (4 selections with fitness functions and 2 operators) a general evolutionary loop can be constructed that can be seen in Figure 1. In this figure circles represent populations, that is sets of individuals. Squares are selections and operators, which take a set of individuals and return a new set of individuals. These individuals may be only filtered (selections) or newly generated (operators). The selections have a goodness measure, that is a fitness function, according to which they filter the populations. These fitness functions are F_{ps}, F_{cp}, F_{el} and F_{ss} for the different selections.

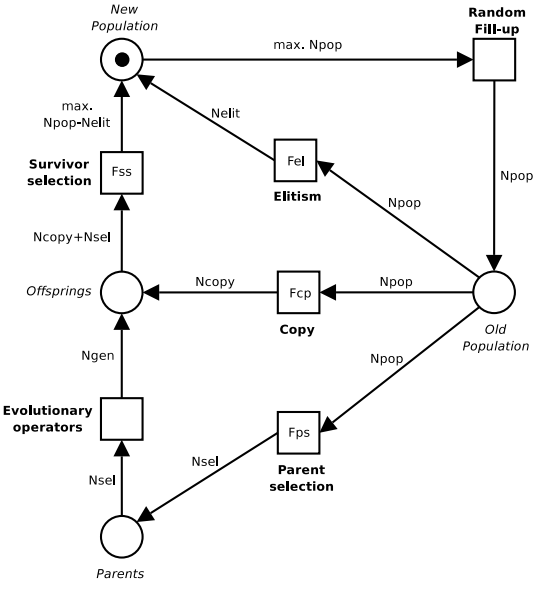


Fig. 1. General EA loop

Before the algorithm starts, several constants are defined. These are N_{pop}, N_{sel}, N_{gen}, N_{copy} and N_{elit}, showing the size of the population, the number of selected, generated and copied individuals and the size of the elitism respectively. The loop begins with a new population marked with a dot. If there are less than N_{pop} elements, the population is filled up with randomly generated individuals. Parent selection using fitness function F_{ps} selects N_{sel} individuals. Then the evolutionary operators are applied on these individuals and N_{gen} new individuals are created, and inserted into the offspring population. From the original population N_{copy} elements are added to this population using copy selection with fitness function F_{cp}. The survivors are selected from the offspring population using survivor selection with fitness function F_{ss} into the new population. Meanwhile N_{elit} individuals are inserted directly from the old population into the new population by the elitism using fitness function F_{el}. This loop can be implemented as Algorithm 1 shows.

Algorithm 1 General EA loop

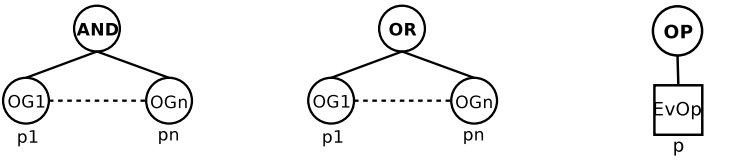
```
GENERAL_EA_LOOP(new_pop)
 1  while new_pop.size < N_pop
 2  do new_pop.insert(Random_Individual())
 3  old_pop ← new_pop
 4  parents ← parent_sel.select(old_pop, N_psel)
 5  offsprings ← operators.apply(parents, N_gen)
 6  offsprings += copy_sel.select(old_pop, N_copy)
 7  N_surv ← min{N_copy + N_sel, N_pop − N_elit}
 8  new_pop ← surv_sel.select(offsprings, N_surv)
 9  new_pop += elitism.select(old_pop, N_elit)
10  return new_pop
```

The experienced reader may have recognized that this model does not consider the phenotype and the genotype. This model simply forgets the phenotype. It is only needed for fitness evaluation, but the fitness function can be considered as a composition of the genotype decoding function and the phenotype evaluation function, thus the phenotype never appears in the computer point of view.

2.1 Generalized Evolutionary Operator

In a common evolutionary algorithm it is usual that first recombination is applied on the population and then mutation is applied on some individuals. This can be generalized, and more operators may be allowed. That is several operators can be applied on the population sequentially. However, one may have more mutations, and want to choose between them. Thus not only a sequential, but also a parallel application has to be allowed. By generalizing this idea the *operator groups* can be defined. An operator group can be a single operator or a set of operators to be applied sequentially or parallelly. So the operators and operator groups can

be organized into a tree-like structure, where the leafs are operators and the internal nodes are operator groups. The three types of groups can be seen in Figure 2 and are the following



(a) and (b) or (c) operator

Fig. 2. Types of operator groups

- AND – the operator groups are applied sequentially from 1 to n on the whole population, using probability p_i.
- OR – the operator groups are applied parallelly on the population, and the result is the union of the resulted populations. The probability of inserting an individual created by group i into the result is p_i.
- OP – the evolutionary operator is applied on the whole population.

The apply method of the operator group takes two populations – an input and an output population – and a size, that is the number of individuals to be generated. The output population does not have to be empty, the new objects are inserted to the end of it. The AND group takes the output of an operator group as the input of the next one. The input of the first one is the input population, and the output of the last one is inserted into the output population, as it can be seen in Algorithm 2.

Algorithm 2 Applying an AND group

```
APPLY(input_population, output_population, size)
1  temporary_population1 ← input_population
2  temporary_population2 ← empty_population()
3  for i ← 1 to n
4  do og[i].apply(temporary_population1, temporary_population2, size)
5     temporary_population1 ← temporary_population2
6     temporary_population2 ← empty_population()
7  op.merge_with(temporary_population1)
```

The OR group first partitions the required size into n partitions using the probabilities of the operator groups. Then it uses the operator groups to create individuals according to the partitions. The created individuals are inserted directly into the output population. This method can be seen in Algorithm 3.

Algorithm 3 Applying an OR group

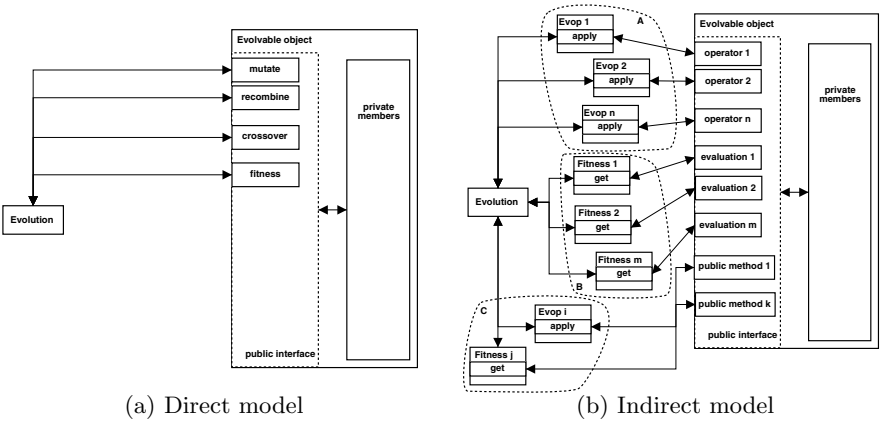
APPLY($input\_population, output\_population, size$)
1 $sizes \leftarrow weighed\_partitions(size, probabilities)$
2 **for** $i \leftarrow 1$ **to** n
3 **do** $og[i].apply(input\_population, output\_population, sizes[i])$

3 Object Oriented EA

In this section an object oriented evolutionary algorithm is designed using the model and ideas from the previous section. The central object in the system is of course the evolutionary algorithm itself, but first the most important building block is designed, and it is the individual, that is the representation of the problem.

3.1 Individuals

The most common way to implement individuals is to create an evolvable object, and then use its descendants (subclasses) as representations for the problem, as outlined in Figure 3(a) The Evolvable class gives the common interface, so the



(a) Direct model (b) Indirect model

Fig. 3. Different models for Evolvable objects

evolutionary algorithm can modify the objects, and get their fitnesses. This will be referred to as *direct access*, since the EA knows the interface of the Evolvable class, and directly modifies the object. The direct model is used by the previously mentioned C++ implementations of EA (GAlib, EO, GEA).

However, in this paper another approach is followed. Since it is not known, how many operators the user wants to use, or how the fitness is calculated, the Evolvable subclasses cannot be forced to implement a given number of operators and/or a fitness function. Thus the EA class has no knowledge of the

Evolvable class, these objects cannot communicate directly and therefore an object is needed to connect them. This approach can be seen in Figure 3(b), where the contact between the EA and the individual is established by Evolutionary Operator and Fitness objects.

In this case the Evolvable objects may have several operators and fitness functions, but it is also possible to modify and evaluate them using other, not EA specific public methods. The disadvantage of this approach is that one has to implement small Evolutionary Operator and Fitness Function classes. However, operators have to be implemented anyway, so the only difference is that they are now in separate classes. The advantage is generality. The Evolvable subclass do not have to implement a certain number of operators. It can implement more, less, or even none. The only required methods are randomization, cloning and printing. The first two functions are needed because during the EA new individuals have to be constructed. Another advantage of this approach, is that it is also easier to parameterize the operators, since the Evolutionary Operator classes may have private fields to hold several parameters.

3.2 Population

The individuals make up the population, and it can be implemented as a vector, or as a set using standard arrays or the Standard Template Library. To construct a versatile evolutionary algorithm, it can be very useful to separate the population from the EA, and use an abstract class with a well defined interface. Thus the EA does not have to care what is inside of the population. It may be a simple array, or it may be in a file, in a database or even on a remote machine somewhere in the Internet. What the EA needs is the possibility to apply operators and selections on the population to create new populations. And of course inserting and removing individuals.

3.3 Operators

As mentioned previously, the operators are the link between the EA and the individuals. Furthermore, they can be organized into operator groups. The only functionality needed from operators and from operator groups is that they must be able to be applied on populations. That is operators and operator groups take populations and create new populations.

3.4 Selection

Selections are similar to operators. They also need a fitness function, according to which they filter the individuals. As it can be seen from the general EA model, the EA class does not have to know the fitness function either, only the selection method.

3.5 Algorithm

As discussed previously, the evolutionary algorithm itself knows the populations, an operator group and four selections. Later it will be easier and more efficient, to consider the populations created during an evolution loop as *subpopulations*, that is parts of one population. Thus, the EA object has exactly one Population, one Operator Group, and four Selections. The Population includes Evolvable objects, the Operator Groups are composed of other groups and Operators, and each selection has its own Fitness Function.

4 Implementation of the OOEA

Using the ideas and waypoints mentioned in the previous section, one can implement a general EA. It is not the purpose of this paper to give a detailed description of the implementation, but the important points are discussed in this section. The class diagram of the system can be seen in Figure 4. The classes and their most important functions are the following:

- **EA**: implements an evolutionary algorithm. Contains a Population, Operator Group and four Selections. Furthermore it has 5 parameters, which were given at the general EA model. Besides the functions to set the parameters and other components, it has a method to carry out one or more evolutionary steps.
- **Population**: contains Evolvable objects, and has several set-like operators. To be fast, only a pointer is given back, when an individual is requested, but it is constant for security. The Population has one or more subpopulations numbered from one, which can be used to store temporary populations, like the set of parents, or the set of offsprings.
- **Evolvable**: the individual in an EA. It can be cloned and printed, and it can generate a random individual.
- **SelectionMethod**: the most important method of this class is `apply`, which takes a Population, and inserts the selected individuals into a second Population. Usually it has a Fitness Function.
- **FitnessFunction**: it has a method called `get` that returns the fitness value of an individual or the fitness values for a whole population. There is a special descendant of this class called FunctionFitness. It stores a function pointer, and uses it for fitness evaluation.
- **OperatorGroup**: contains a tree of OperatorGroups and/or Evolutionary Operators. The method `apply_on` is the most commonly used function. It takes a population of parents, applies the group on it as described previously, and inserts the new individuals into another population.
- **Evolutionary Operator**: the evolutionary operator. Its method `apply` is the counterpart of the `apply_on` method for operator groups.

These classes can be divided into three groups. EA and OperatorGroup are *fixed classes* for any evolutionary algorithm, the user need not and can not modify them. The Population and the SelectionMethod classes are *customizable classes*,

they may be reimplemented, but it is not necessary, if the user does not need special subclasses. There are already several selection and population implementations, they can be used with many problems. Evolvable, EvolutionaryOperator and FitnessFunction are *problem specific classes*, thus in many cases they have to be implemented by the user. However, bitstrings and real vectors, and the appropriate operators are already implemented.

5 Examples

Using this general implementation many types of evolutionary algorithms can be implemented. Some examples are listed in the following.

- *ES with , or + strategy*: real vector implementation can be used, and in case of + strategy the copy selection can be used to insert the parents into the offspring population.
- *Traditional GA*: the bitstring is already implemented in the system with simple operators.
- *Island Model*: an evolvable population may be implemented as a subclass of classes Evolvable and Population, and the fitness evaluation can include an evolutionary algorithm.
- *Meta-ES*: similar to the Island Model
- *Parallel EA*: more populations and more EA objects with the same parameters may be created to implement this.
- *Distributed EA*: using a derivative of the Population object the individuals may be stored on different machines. It is also possible to distribute the individuals only for fitness evaluation, by implementing an appropriate `get` method.
- *Co-evolution*: the second `get` method of the fitness function takes the whole population, so it can be used to evaluate the individuals during a co-evolution.

5.1 Using the Library

When the representation form is already implemented, to use evolutionary algorithms is very easy. With 100-200 lines of C++ code even complicated problems can be solved. An example is detailed in the following.

1. Implement a fitness function

```
double fitness(const Evolvable &i){ /* fitness calculation */ }
```

2. Create a new population then a random individual, and insert this individual into the population

```
void test_ea(){
  ArrayPopulation *pop = new ArrayPopulation();
  ArrayPopulation results;
  EvolvableRealVector evreal(4);
  pop->insert(evreal.random());
```

3. Create a fitness function object

```
FunctionFitness *ff = new FunctionFitness();
ff->set_func(fitness);
```

4. Create the used selection methods, and set their fitness function, if necessary

```
SequentialSelection *copy = new SequentialSelection();
BestSelection *bs = new BestSelection(500);
FitPropSelection *fps = new FitPropSelection(500);
bs->set_fitness(*ff); fps->set_fitness(*ff);
```

5. Create the operator objects

```
EvOpRealMutate eo_mut;
EvOpRealRecombine eo_rec(false);
```

6. Combine the operator objects into an operator group

```
OperatorGroup og, rec, mut;
rec.set_op(&eo_rec); rec.set_prob(800);
mut.set_op(&eo_mut); mut.set_prob(500);
og.set_and(rec); og.add_group(mut); og.set_prob(1000);
```

7. Create an evolutionary algorithm by giving the parameters. Then set the population, the selections and the operator

```
EA *myEA=new EA();
myEA->set_params(1000,50,10,10,980);
myEA->set_population(pop);
myEA->set_psel(fps);
myEA->set_copy(bs); myEA->set_elitism(bs);
myEA->set_ssel(copy);
myEA->set_operator(&og);
```

8. Run the EA! After each step you can get individuals from the population, read their fitnesses, print them, examine halting criteria, and so on

```
for (int i=0; i<100; ++i){ myEA->step(); /* save, print */ }
```

9. At the end delete the objects

```
delete myEA; delete fps; delete bs; delete copy; delete pop;
}
```

5.2 Extending the Library

It is easy to extend the library, only the following points have to be followed.

– A subclass of class Evolvable has to be implemented with the `clone`, `random` and `print_on` functions. The class may contain special methods for evolutionary operators or for fitness evaluation.

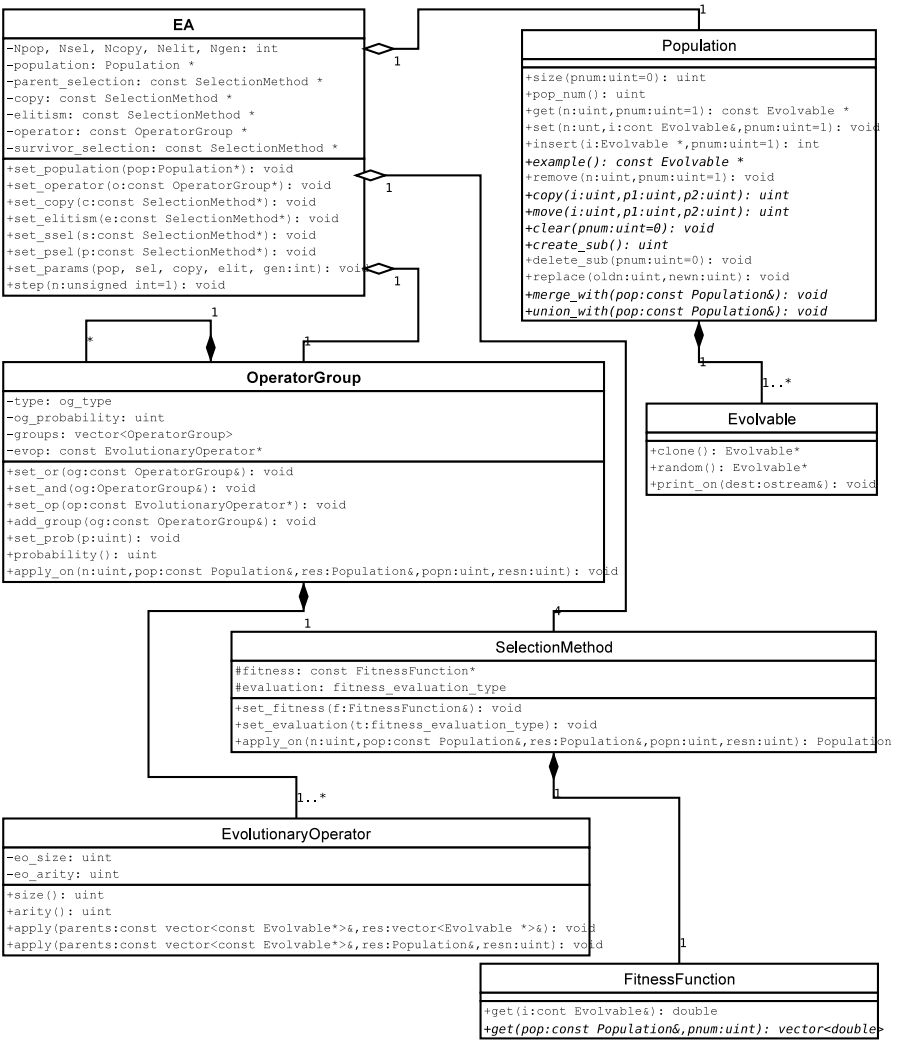


Fig. 4. UML class diagram of the general EA library

- The evolutionary operators have to be implemented. The most important function is `apply`, which can use the public methods of the Evolvable class.
- The fitness function has to be implemented. It can be either a subclass of FitnessFunction, or just a function as shown in the example.
- For special needs new selection methods may be implemented or the Population class may be overloaded.

6 Conclusion, Future Work

In this paper a general evolutionary algorithm was designed, and the steps of its implementation were shown. Using this general EA many special types of evolutionary algorithms can be implemented. The constructed system is simple and still versatile, easy to use and easy to extend. It was not among the aims of this paper to compare different programming libraries. However, it is planned to compare some of them with respect to CPU and memory usage.

Some features would be nice to be added to the library like a logging function, or error checking. It is also planned to change the randomization to be similar to the evolutionary operator, that is separated from the Evolvable class. When this is done, it may be considered to replace the inheritance based design to a template based one. This would mean even higher level of generality.

References

[eaL] eaLib – a Java Evolutionary Computation Toolkit. Technical report, Technical University Ilmenau, Germany. http://www.evolvica.org/ealib/index.html.

[Evo] Evolvica – Evolutionary Algorithms Framework in Java. Technical report, Technical University Ilmenau, Germany. http://www.evolvica.org/.

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

[Gre84] J J Grefenstette. Genesis: A system for using genetic search procedures. In *Proceedings of a Conference on Intelligent Systems and Machines*, pages 161–165, 1984.

[Hol75] John H. Holland. *Adaption of Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, Michigan, 1975.

[Koz92] John R Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, Massachusetts, 1992.

[Mer] J J Merelo. Eo evolutionary computation framework. http://eodev.sourceforge.net/.

[Rec73] Ingo Rechenberg. *Evolutionsstrategien: Optimierung Technischer Systeme nach Prinzipen der Biologischen Evolution*. Fromman-Holzboog, Stuttgart, 1973.

[SGE94] R. Smith, D. Goldberg, and J. Earickson. Sga-c: A c-language implementation of a simple genetic algorithm, 1994.

[Tót00] Zoltán Tóth. *Generic Evolutionary Algorithms Programming Library*. Master's thesis, University of Szeged, Hungary, 2000.

[Wal96] M Wall. Galib: A C++ library of genetic algorithm components. Technical report, Massachusetts Institute of Technology, Mechanical Engineering Department, April 1996. http://lancet.mit.edu/ga/.