# SYSGP – A C++ library of different GP variants

Markus Brameier Wolfgang Kantschik Peter Dittrich

Wolfgang Banzhaf

Fachbereich Informatik
Universität Dortmund
44221 Dortmund, GERMANY
email: brameier,wkantsch,dittrich,banzhaf@ls11.informatik.uni-dortmund.de

**Abstract**

In recent years different variants of genetic programming (GP) have emerged all following the basic idea of GP, the automatic evolution of computer programs. Today, three basic forms of representation for genetic programs are used, namely tree, graph and linear structures. We introduce a multi-representation system, SYSGP, that allows researchers to experiment with different representations with only a minimum implementation overhead. The system further offers the possibility to combine modules of different representation forms into one genetic program. SYSGP has been implemented as a C++ library using templates that operate with a generic data type.

# 1 Introduction

Genetic programming (GP) has been formulated by Koza originally as an evolutionary method for breeding computer programs in form of tree structures [5]. By applying mechanisms of natural evolution GP searches for the best program that represents a solution to a given problem. The major differences compared to other evolutionary methods like genetic algorithms [4] or evolution strategies [8] are that the genetic material in GP is executable and variable in size.

In the meantime, genetic programming has been applied to real problems in industry, finance, medicine, robotic control, image processing, speech recognition, and in other disciplines. Most applications demonstrate the abilities of GP in data mining, the discovery of regularities within large data domains, by evolving classifiers or regression functions with sufficient generalization quality. Comparison to neural networks shows that GP displays a similar performance as neural nets, even with a relatively small number of generations [2].

In recent years, two new generic representations for genetic programs have been introduced into the area of genetic programming, *graph* and *linear* structure. It became clear that the methods of GP are not confined to the world of parse trees but can be applied to a variety of program representations with much success. For details of the present state of genetic programming, or for a more thorough introduction than Section 2 can provide, the reader is referred to our text book [1].

In contrast to this variety of GP, however, most publicly available GP systems deal with a single program representation only. SYSGP, the system introduced here, includes all three generic representation forms, i.e. trees, graphs and linear structures, such that it is possible now to apply different program representations to the same problem and to compare the results with a minimum of implementation overhead. Experimenting with different program representations is reasonable since for each representation form there exist problem domains that are more suitable than others.

Another feature of SYSGP is the potential for combination of different GP variants. A genetic program may consist of one or more main modules each using a local set of submodules. Main modules and submodules may belong to different representations. Thus, several types of combination are possible to choose from. The motivation for mixed representations is in exploiting the advantages of two or more representation forms for certain applications.

SYSGP is a library of several C++ classes that offers all the elements of a genetic programming system. The library employs the object oriented programming scheme. By inheriting new classes and overloading of existing functions it is easy to adapt the system to a specific application. A template-based implementation supports all types of data the genetic programs may operate with. The definition of data types and data manipulating functions is common for all three genetic representations. This should reduce the effort necessary to establish the same problem with different data types or representation types.

SYSGP has been developed at the University of Dortmund, Department of Computer Science, Chair of SYStem Analysis and is currently available on request from the authors. It is intended to become a public domain tool in the near future.

## 2 Variants of Genetic Programming

*Evolutionary algorithms* imitate mechanisms of natural evolution to optimize a solution towards a predefined goal. *Genetic programming* is the newest of four basic research subfields of evolutionary algorithms further including *genetic algorithms* [4], *evolution strategies* [8] and *evolutionary programming* [3]. A general evolutionary algorithm may look like:

1. Create an initial population of individual solutions randomly.
2. Select individuals from the population based on a selection policy and compare them with respect to a fitness measure.
3. Modify only fitter individuals by the following genetic operators:
   - Identical *reproduction*
   - Exchange of a single unit in an individual (*mutation*)
   - Exchange of substructures between two individuals (*recombination*)
4. The individual with the best fitness represents the best solution found so far.

*Genetic programming* (GP) operates on computer programs as individuals. In contrast to other evolutionary methods, the individuals are of variable shape and size. Following Darwin's principle of natural selection the evolutionary process searches for a program that fits a set of *fitness cases* (i.e. input-output examples) best. The *fitness* of a genetic program derives from the error between the given outputs and the outputs predicted by the program.

The GP approach has been formulated originally using *tree* structures to represent the evolved programs. In recent years, the area of genetic programming has expanded considerably and now includes *linear* and *graph* representations as well. Each type of program representation differs in execution order, use of memory, and genetic operators. All these variants work without changing the basic idea of genetic programming, i.e. the evolution of computer programs. In the following subsections we describe the three GP representations in some more detail and in Section 3 we present some possible combinations of these representations.

### 2.1 Tree Representation

The earliest and most commonly used approach to genetic programming is the evolution of tree structures represented by variable length expressions from a functional programming language like LISP [5]. The inner nodes of these trees are functions that may have side-effects while the leafs are terminals that represent input variables, constants or functions without parameters.

During evolution genetic operators, i.e. crossover and mutation, transform the genetic material in the population. The operators must guarantee syntactic closure, i.e. syntactically incorrect programs are not allowed to be generated. Figure 1 illustrates representation and recombination in *tree-based* GP. In each parent individual the crossover operator chooses a node randomly and exchanges the two corresponding subtrees. In general, the crossover points might be directed to inner nodes with higher probability than to terminal nodes. In SYSGP these probabilities can be adjusted respectively. The mutation operator exchanges constants, variables or functions with certain rates. Functions may be replaced either with functions expecting the same number of parameters only or with any function from the function set. In the second case subtrees need to be deleted or created respectively. The exchange of terminals for functions and vice versa is allowed optionally.

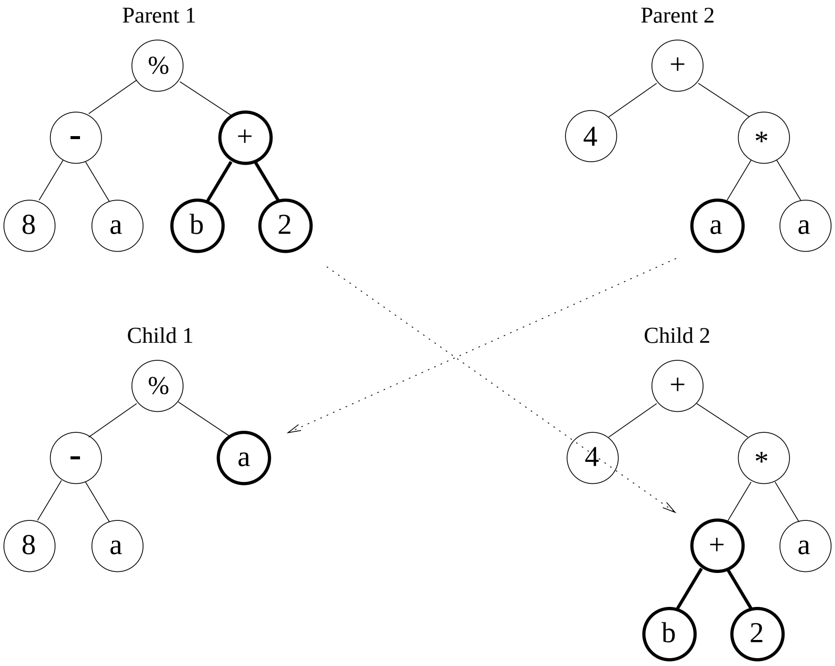


Figure 1: Program structure and crossover in tree-based GP

### 2.2 Linear Representation

In *linear* GP the representation of the programs is a linear sequence of instructions. Its main characteristic in comparison to tree-based GP is that what is evolved are programs of an imperative language (like C), not expressions of a functional programming language (like LISP).

A radical form of linear GP, the evolution of machine code, has been introduced in [6] and further developed in [7]. This method, now called AIMGP (for *Automatic Induction of Machine code by Genetic Programming*) [1], uses machine code as individual programs which are manipulated as bit sequences in memory and directly executed without passing through an interpreter during fitness calculation. This approach results in a significant speedup compared to interpreting systems.

The implementation of linear GP in the SYSGP library represents an individual program as a variable length list of C instructions that operate on (indexed) variables or constants. In linear GP all operations, e.g. $a = b * 2$, implicitly include an assignment to a variable. After a program has been executed its output value(s) are stored in designated variables. This is in contrast to tree-based GP where assignments and multiple outputs had to be incorporated explicitly by using an extra indexed memory and special functions for its manipulation. The number of allowed operands is not necessarily two and can be adjusted. In addition, conditional branch instructions exist that skip a variable (mutable) number of subsequent instructions if the condition is false.

Figure 2 illustrates the two-point crossover used in linear GP for recombining two parent individuals. A segment of random position and length is selected in each of the two parents for the exchange. If one of the children would exceed the maximum length, crossover with equally sized segments is performed. Crossover points only occur *between* instructions but not within. *Inside* the instructions the mutation operation ensures that only instructions with valid operators and valid ranges of variable indices and constants are created. Operators, variables and constants are exchanged with separate and user controlled rates. Note that exchange of a variable by a single mutation may have a significant effect on the program flow in linear GP [2].

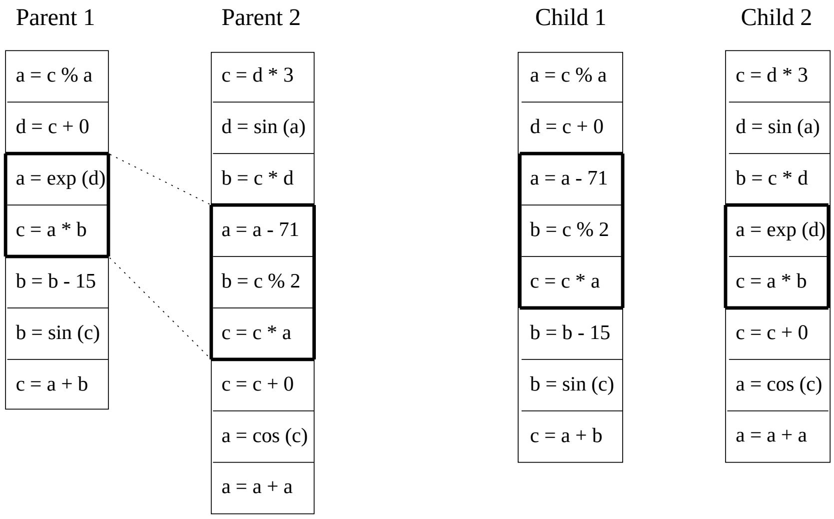


Figure 2: Program structure and crossover in linear GP

### 2.3 Graph Representation

The graph structure is the newest program representation used in GP and has been introduced in [9]. In *graph-based* GP each program is constructed as an arbitrary directed graph of nodes, an indexed memory for (input) variables, and a stack (see figure 3). As an arbitrary directed graph of $N$ nodes, each node can have as many as $N$ outgoing edges. But the graph structure is more than nodes connected by edges indicating the possible flow of control.

Each node in the program has two parts, *action* and *branching decision*. The *action* part is either a constant or a function which will be executed when the node is reached during the interpretation of the program. Data is transferred among the nodes by means of a stack. An action function gets its inputs from (the top of) the stack and puts its output onto the stack again. After the action of a node is executed, an outgoing edge is selected according to the branching decision. This decision is made by a *branching function* that determines the edge to the next node, while using the information held on the top of the stack, in memory or in a special *branching constant*. Hence, not all nodes of a graph are necessarily visited during an interpretation.

Each program has two special nodes, a *start* and a *stop node*. The start node is always the first node to be executed when a program begins. When the stop node is reached, its action is executed and the program halts. Since the graph structure inherently allows loops and recursion, it is possible that the stop node is never reached during the interpretion. In order to avoid that a program runs forever it is terminated after a certain *time threshold* is reached. In SYSGP the time threshold is implemented as a fixed maximum number of nodes which can be executed in a program. If a program stops, its outputs are the current values residing in defined memory locations or on the stack.

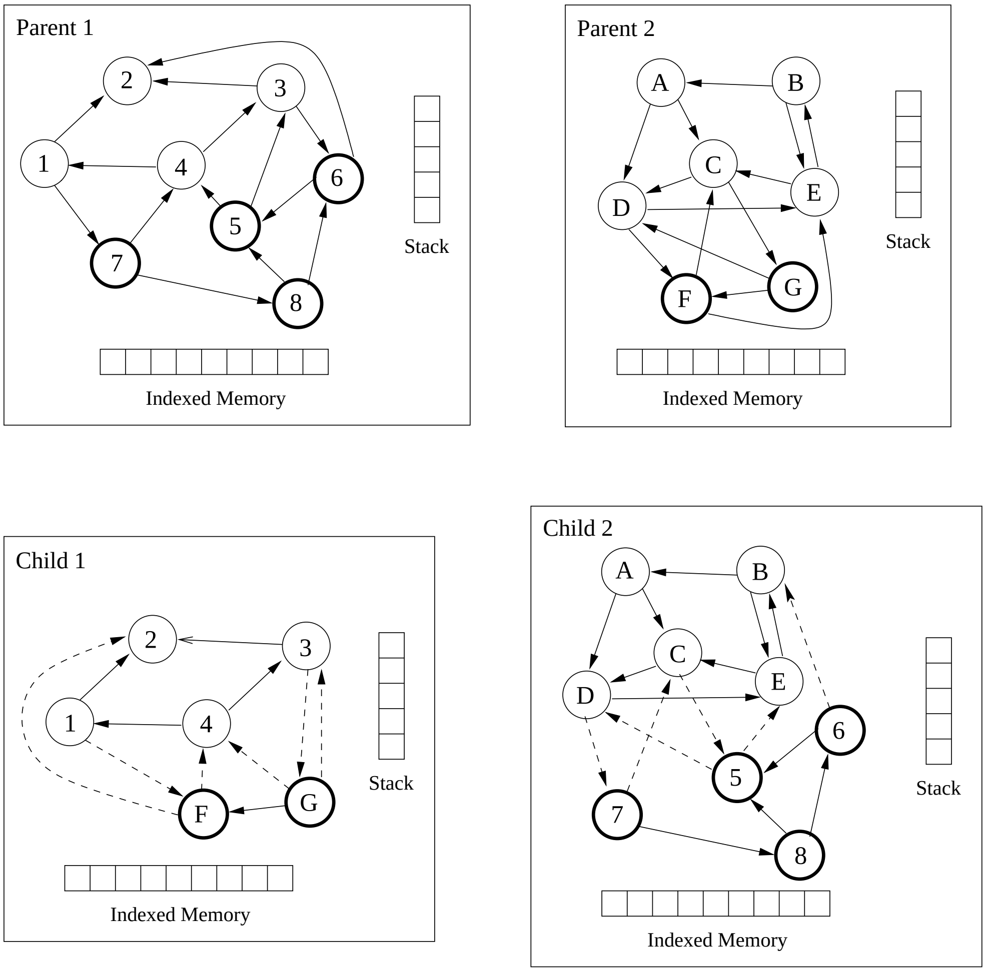


Figure 3: Program structure and crossover in graph-based GP

The crossover operator combines the genetic material of two parent programs by swapping certain program parts. In graph-based GP crossover is more complicated than a simple exchange of single nodes. The following algorithm [9] is applied for recombination here:

1. Divide each parent graph into two node sets randomly.
   - Label all edges *internal* if they point to another node in the same set, or *external* otherwise.
   - Label nodes in each set as *output* if they are the source of an external edge and as *intput* if they are the destination of an external edge.

2. Exchange one set from each graph.
3. Recombine so that all *external* edges in each fragment point to randomly selected *input* nodes in the other.

This method assures that all edges are connected in the two child graphs and that valid graphs are generated. Figure 3 shows a crossover example with this method.

The mutation operator selects a subset of nodes randomly and changes either action, branching decision, or outgrade with certain probabilities. The outgrade of a node is modified by adding or removing a single outgoing edge.

## 3 Combination of different Representations

Using subprograms in genetic programming allows problems to be solved hierarchically. A complex problem may be decomposed into simpler subproblems in such a way that the overall solution is a combination of subsolutions that are reused several times. This can result in simpler and less complex solutions.

An individual in SYSGP may be composed out of one or more main programs each using a variable number of local subprograms (ADFs [5]). Main program(s) and subprograms are evolved in parallel while each module is built from a separate set of functions, variables and constants. The function set of a main program includes identifiers of all its subprograms. For each module the *representation type* (see Section 2) can be choosen *arbitrarily* in SYSGP. As a result many *individual types* of configuration are possible. Figure 4 gives an example of an individual combining three representations.

A generic interpreter identifies the (representation) type of each component module in an individual and invokes the interpreter of the respective representation. The interpretation of an individual is reduced to the interpretation of its main program(s) while the interpretation of a main program branches to a subprogram if invoked by the main program. For all three representation types the interpretation of a subprogram works locally. The main program is only affected by the exchange of the input and output values just as executing a "normal" function from its function set (see Section 2). When the interpretation of a subprogram is terminated, the interpretation of the main program continues at the position where the execution of the subprogram had been initiated. If more than one main program is used the results of the overall interpretation are stored in an output vector.

Furthermore, a generic crossover operator and a generic mutation operator exist for individuals, both invoking the operators of the respective module representations. The crossover operator performs a variable number of crossover operations between two component modules of the same position. (Each module is given a fixed position index in an individual.) For both, main modules and submodules, the probability of crossover can be adjusted separately. The mutation operator changes a variable number of main programs and a variable number of subprograms with certain probabilities as well.

The benefit of mixed representations in general is that they allow to combine the advantages of several representation forms. In this context the combinations discussed above allow subproblems to be solved by modules of another representation as is used for the main program.

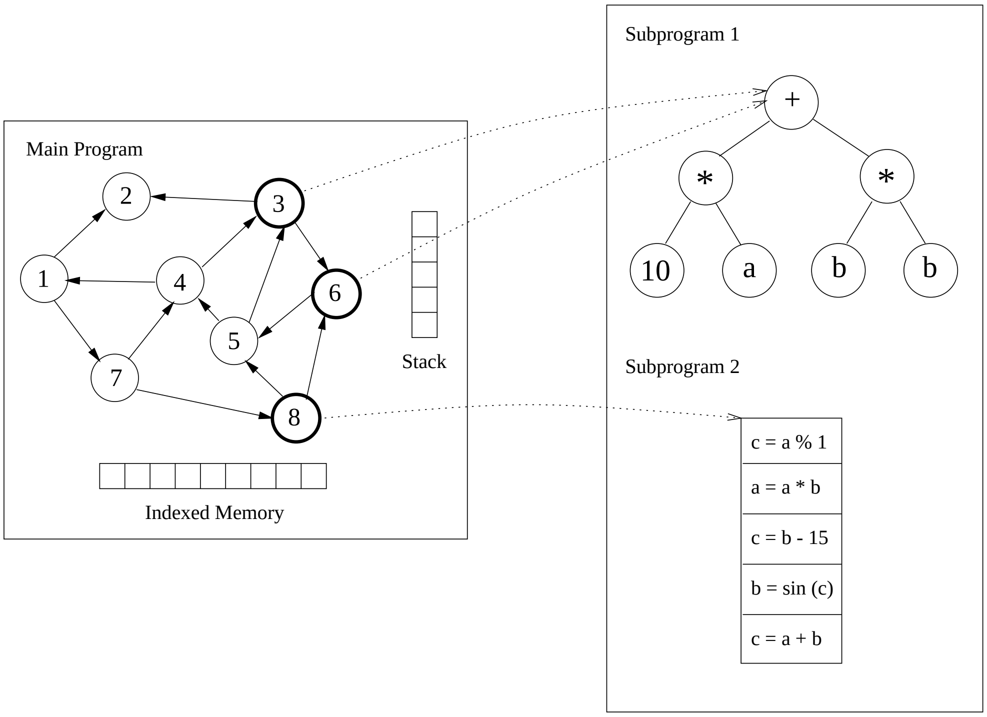


Figure 4: Example of an individual with mixed representation

# 4 Implementation

SYSGP requires the user to create his/her own GP system using the components offered by the numerous classes and functions of the library. In this way especially the evolutionary algorithm and the fitness function offer high flexibility.

All classes and global functions of the SYSGP library are templates operating with a generic data type ($T$) which is defined by the user. This abstraction allows *any* data type to be used for variables and constants in genetic programs. Further, the function set can be defined *independently* from the representation type, i.e. there is only one set used by all representations. The common function set that is predefined in SYSGP includes arithmetic operators, conditional branches, trigonometric and exponential functions.

```
template <class T> add(SYSGP_Interpret<T>);

SYSGP_Op<T> Add(add, "+", Min, Max);
```

The preceding code gives an example for a function definition in SYSGP. All functions have to return an object of type $T$ and expect a (representation specific) interpreter of class *SYSGP_Interpret* as parameter. Additionally, an operator object of type *SYSGP_Op* has to be created for every function holding the function itself (`add` here), its symbol name and the minimum and maximum number of parameters allowed. This object is inserted into the function set *SYSGP_OpSet*.

A graphical user interface (GUI) has been developed for SYSGP in order to make the initialization and controll of runs more comfortable. An integrated visualization component documents statistical results graphically during runtime. Some further features of SYSGP that have not been mentioned yet include:

- 4 selection methods: fitness proportional selection, tournament selection, $(\mu, \lambda)$ strategy and $(\mu + \lambda)$ strategy (adopted from evolution strategies [8])
- steady state GP (optional)
- multiple populations (demes) with variable topology and migration strategy
- loading and saving of runs
- comfortable management of system parameters
- tools for statistical analysis

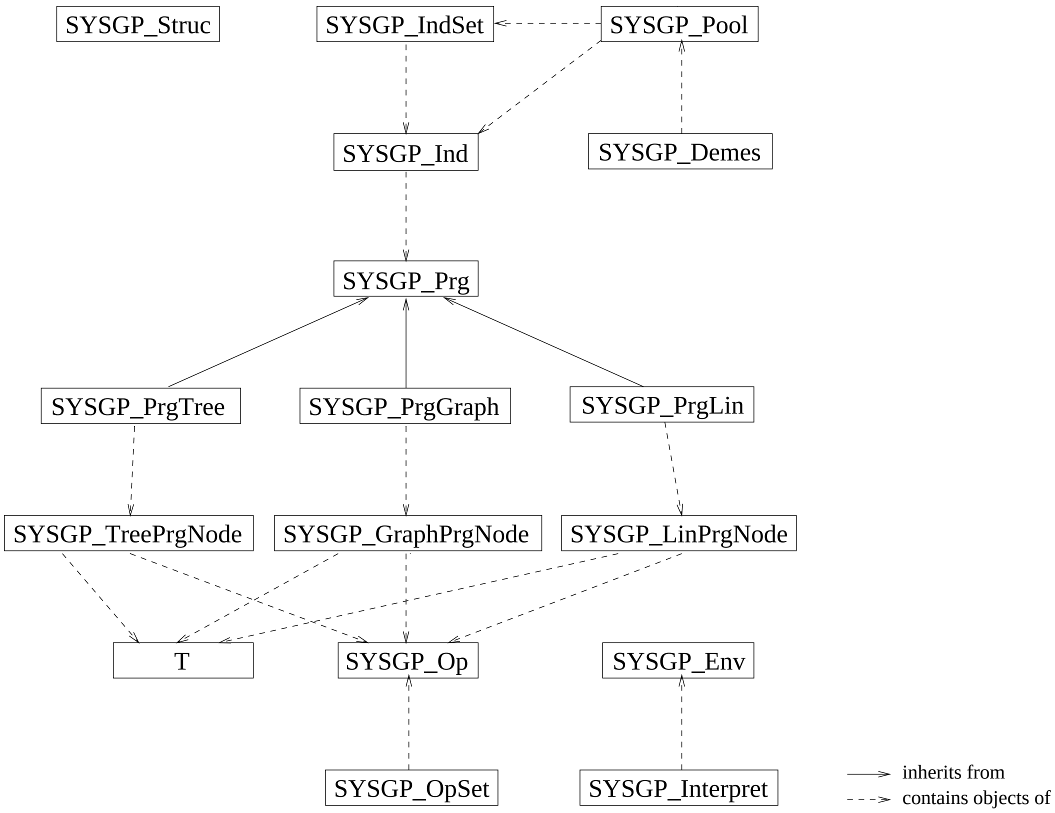


Figure 5: Class hierarchy of the SYSGP library

Figure 5 shows the class hierarchy as implemented in the SYSGP library. Compatibility to other libraries is granted by a global name space concept that encapsulates all global definitions through a common name prefix (*SYSGP* here). The population class *SYSGP_Pool* is a template class holding objects of any type (here *SYSGP_Ind* or *SYSGP_IndSet*). Individual objects of type *SYSGP_Ind* are composed of objects from the general representation class *SYSGP_Prg* or from any inherited representation class. An *individual* includes *one* main program with local subprograms of arbitrary representation. Class *SYSGP_IndSet* combines several individual objects into one *super individual*. All node classes (*SYSGP_XPrgNode*) either contain an instance of the operator class *SYSGP_Op* or the data class *T*. The stack and/or the indexed memory used by the different interpreters are hold in the enviroment class *SYSGP_Env*. Finally, class *SYSGP_Struc* stores all controll parameters of the system including common and representation specific parameters.

# 5 An Example

Each representation form has different advantages and problems exist for which it is better suited. In order to demonstrate that it is reasonable to experiment with different representations we apply all three GP variants to the same problem here, a symbolic regression of the *sinus* function. In general, symbolic regression problems deal with the approximation of a set of $n$ numeric input-output relations $(x, y)$ by a symbolic function. The evolved programs should produce an output as close as possible to $y$ if the input is $x$. The *fitness* of an individual program $p$ can be defined here as the square error between all given outputs $y$ (here $y = sin(x)$) and the predicted outputs $p(x)$:

$$fitness(p) = \sum_{i=1}^{n} (p(x_i) - sin(x_i))^2$$

The *generalization quality* of a genetic program depends on its ability to induce a continuous function from the $n$ *fitness cases* (*training set*). Generalization is monitored during the evolutionary process by testing the currently best individual on a set of $n$ *unknown* input-output examples (*validation set*). On our sample runs, training and validation set hold 50 uniformly distributed examples with input range $[0,2\pi]$ each.

All variants of GP have been configured without using subprograms, with a population size of 1000 individual programs and a maximum crossover and mutation rate of 100%. The common function set holds arithmetic operators $(+ \; - \; * \; /)$ only.

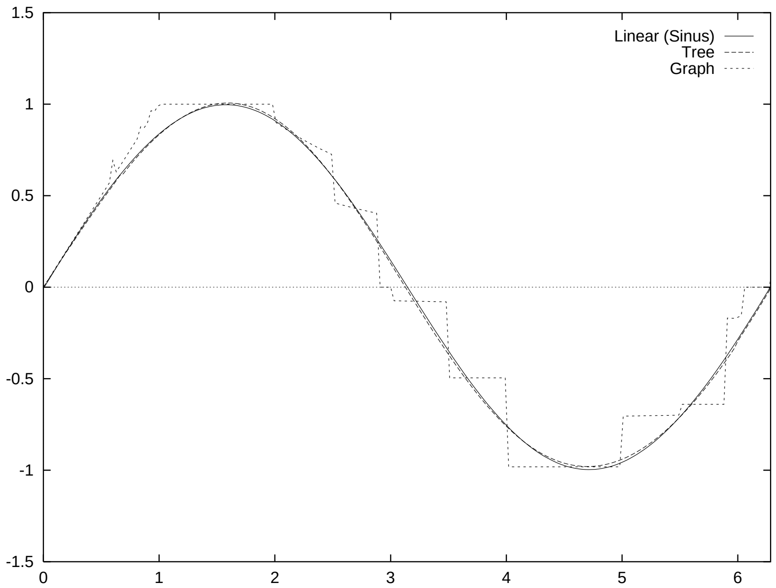


Figure 6: Best approximation found by each variant.

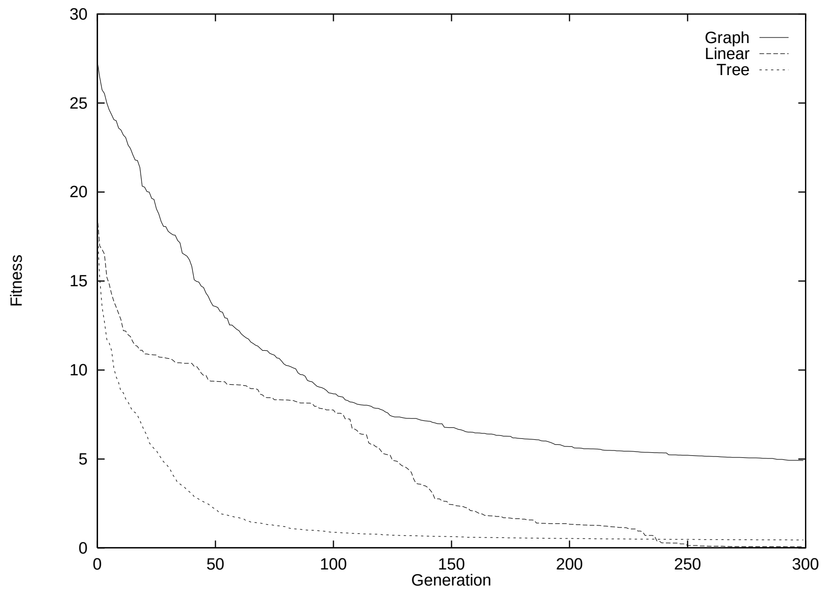


Figure 7: Best fitness of each variant

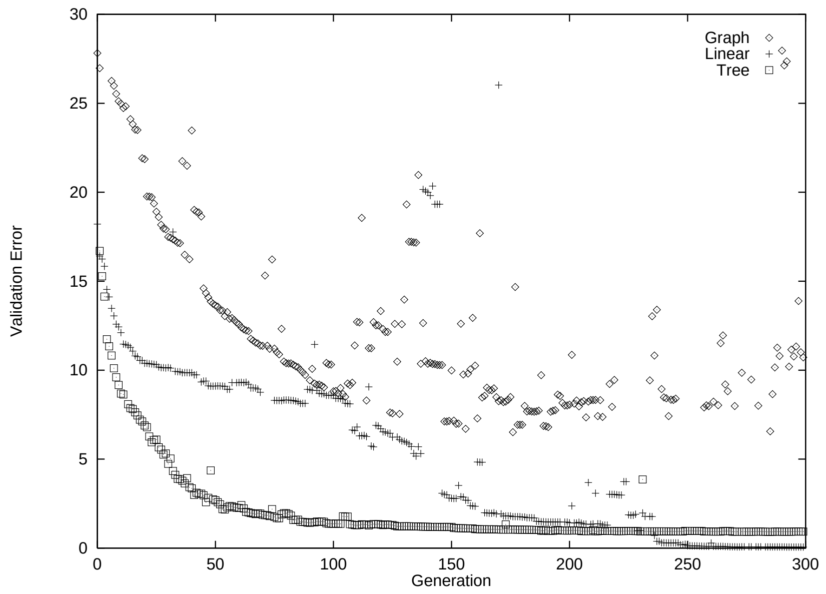


Figure 8: Best validation error of each variant

Figures 7 and 8 show fitness progress and generalization (validation) error for the best individuals over time (in generations). For each representation the results of 20 runs have been averaged. One can see that linear and tree-based GP outperform graph-based GP in fitness and generalization quality for the specific regression problem under consideration. Graph-based GP is generally more suitable for classifications than for regression tasks. After 300 generations best fitness and validation error come closer to zero for linear GP than for tree-based GP. Actually, best fitness zero has been observed several times with linear GP but never with tree-based GP. In contrast to that tree-based GP runs converge faster at the beginning. It is further interesting to note that the progress in tree-based and graph-based GP is rather continuous while linear GP processes more stepwise.

Figure 6 graphically shows a comparison of the best approximations found by the different GP variants for 200 *unknown test inputs*. The best result for graph-based GP differs significantly from the optimum while tree-based GP comes very close to the real sinus function. Linear GP has even been so successfull here that there is no difference from the optimum visible.

We would like to point out that this simple demonstration is not intended to argue for or against a certain kind of genetic programming. It just should give a motivation for experimenting with different variants when applying GP to a certain problem.

## 6 Summary and Future Work

We have introduced a multi-representation system that incorporates three variants of genetic programming – tree-based, graph-based and linear GP. We have described the different GP representations and possible forms of combination in an individual. Finally, all main GP variants have been demonstrated using the same example problem.

The most significant features of SYSGP can be summarized as follows:

- 3 basic representations for genetic programs
- combination of different representations
- generic data type
- general definition of the function set

The class structure of SYSGP allows new representations to be added easily. In future versions other evolutionary methods like genetic algorithms or evolution strategies might be integrated. It is expected that a simultaneous evolution of the vector representation can be useful for certain applications, especially if parameters are to be optimized during the evolutinary process.

## Acknowledgements

This research was supported by the Deutsche Forschungsgemeinschaft, Sonderforschungsbereich 531, project B2.

## References

[1] W. Banzhaf, P. Nordin, R. Keller and F. Francone (1998) *Genetic Programming — An Introduction. On the automatic Evolution of Computer Programs and its Application.* dpunkt/Morgan Kaufmann, Heidelberg/San Francisco.

[2] M. Brameier and W. Banzhaf (1998) *A Comparison of Genetic Programming and Neural Networks in Medical Data Analysis.* Technical Report, University of Dortmund, Reihe Computational Intelligence, Sonderforschungsbereich 531.

[3] L.J. Fogel, A.J. Owens and M.J. Walsh (1966) *Artificial Intelligence through Simulated Evolution.* Wiley, New York.

[4] J. Holland (1975) *Adaption in Natural and Artificial Systems.* University of Michigan Press, Ann Arbor, MI.

[5] J. Koza (1992) *Genetic Programming.* MIT Press, Cambridge, MA.

[6] P. Nordin (1994) *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code.* In K.E. Kinnear, editor, *Advances in Genetic Programming*, MIT Press.

[7] P. Nordin and W. Banzhaf (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In: *Proceedings of Sixth International Conference of Genetic Algorithms, Pittsburgh, 1995.* L. Eshelman (ed.), Morgan Kaufmann, San Mateo, CA.

[8] H.P. Schwefel (1995) *Evolution and Optimum Seeking.* Wiley, New York.

[9] A. Teller (1996) *PADO: A New Learning Architecture for Object Recognition.* In *Symbolic Visual Learning*, Oxford University Press.