A Comparison of Linear Genetic Programming and Neural Networks in Medical Data Mining

Markus Brameier Wolfgang Banzhaf

Fachbereich Informatik

Universität Dortmund

44221 Dortmund, GERMANY

email: brameier,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

We apply linear genetic programming to several diagnosis problems in medicine. An efficient algorithm is presented that eliminates intron code in linear genetic programs. This results in a significant speedup which is especially interesting when operating with complex datasets as they are occuring in real-world applications like medicine. We compare our results to those obtained with neural networks and argue that genetic programming is able to show similar performance in classification and generalization even within a relatively small number of generations.

1 Introduction

Genetic programming (GP) has been formulated originally as an evolutionary method for breeding programs using expressions from the functional programming language LISP [15]. We employ a variant of GP, linear GP, using sequences of instructions of an imperative programming language. More specifically, the method operates on genetic programs being represented as linear sequences of C instructions. One strength of our LGP system is that introns, i.e., dispensible instructions that do not affect program behavior, can be removed before a genetic program is executed during the fitness calculation. This does not cause any changes to the individuals in the population during evolution or in behavior but results in an enormous speedup in execution. Introns are also found in biological genomes, where they appear in DNA of eucaryotic cells. It is interesting to realize that these natural introns are removed, too, before proteins are synthesized. While eliminating introns reduces total runtime a demetic population has been employed to decrease the training time additionally on a generational basis.

Genetic programming and artificial neural networks (NN) in principle can be seen as alternative techniques for the same tasks, including classification and approximation problems. Neural networks increasingly become an alternative to classical statistical methods in the analysis of medical data. Ripley and Ripley [22] review several NN techniques in medicine including methods for diagnosis and prognosis tasks, especially survival analysis. Most applications of neural networks in medicine refer to classification tasks. Only few deal with approximation problems. A comprehensive list of medical applications of neural networks can be found in [4].

In contrast to neural networks, genetic programming has not been used very extensively for medical applications to date. Gray et al. [11] report from an early application of GP in cancer diagnosis where the results had been found to be better than with a neural network. In [16] a grammar-based GP variant is used for knowledge extraction from medical databases. Rules for the diagnosis have been derived from the program tree that uncover relationships among data attributes. The outcomes of different types of classifiers, including neural networks and genetic programs, are combined in [24] which results in an improved prediction of thyroid normal and thyroid carcinoma classes.

In this paper genetic programming is applied to medical data. In particular the linear GP variant is tested on six diagnosis problems that have been taken from the Proben1 benchmark set of real-world problems [20]. The main objective is to show that for the problems discussed genetic programming is able to achieve quite similar classification rates and generalization performance in comparison to neural networks. The application further demonstrates the ability of genetic programming in data mining, where general descriptions of information are to be found in large real-world databases. For supervised learning tasks this normally means to create predictive models, i.e., classifiers or approximators, that generalize from a set of learned data to a set of unknown data.

The paper is organized as follows: In Section 2 the genetic programming paradigm in general and linear GP in particular are introduced. We further present an efficient algorithm that removes intron code from linear genetic programs each time before they are executed. A detailed description of the medical data we have used can be found in Section 3. The setup of all experiments is described in Section 4 while Section 5 presents results concerning intron elimination, classification ability and training time. Finally we discuss some prospects for future research.

2 Genetic Programming

Evolutionary algorithms (EA) mimic aspects of natural evolution to optimize a solution towards a defined goal. Following Darwin's principle of natural selection differential fitness advantages are exploited in a population to lead to better solutions. Different research subfields of evolutionary algorithms have emerged, such as *genetic algorithms* [12], *evolution strategies* [23], and *evolutionary programming* [8]. In recent years these methods have been applied successfully to a wide spectrum of problem domains, especially in optimization. A general evolutionary algorithm can be summarized as follows:

Algorithm 1 (Evolutionary Algorithm):

1. Randomly initialize a population of individual solutions.
2. Randomly select individuals from the population and compare them with respect to their fitness. The *fitness* measure defines the problem the algorithm is expected to solve.
3. Modify fitter individuals using some or all of the following variation operations:
   - *Reproduction.* Copy an individual without change.
   - *Recombination.* Exchange substructures between two individuals.
   - *Mutation.* Exchange a single unit in an individual at a random position.
4. If the termination criterion is not reached, $\rightarrow$ 2.

5. Stop. The best individual represents the best solution found.

A comparatively young and growing research area in the context of evolutionary algorithms is *genetic programming* that uses computer programs as individuals. In an early work Friedberg [9, 10] attempted to solve simple problems by teaching a computer to write computer programs. Cramer [7] was the first who applied an EA for the evolution of computer programs that were represented as variable-length tree structures. Koza further developed this approach and named it "genetic programming" [14, 15].

In GP the individual programs map given input-output examples, called *fitness cases*, while their fitness depends on the mapping error. The inner nodes of the program trees are functions and the leafs are *terminals* that mean input variables or constants. The operators applied to generate individual variants, i.e., recombination and mutation, must guarantee that no syntactically incorrect programs are allowed to be generated during evolution (*syntactic closure*). Figure 1 illustrates the recombination operation in *tree-based* GP.

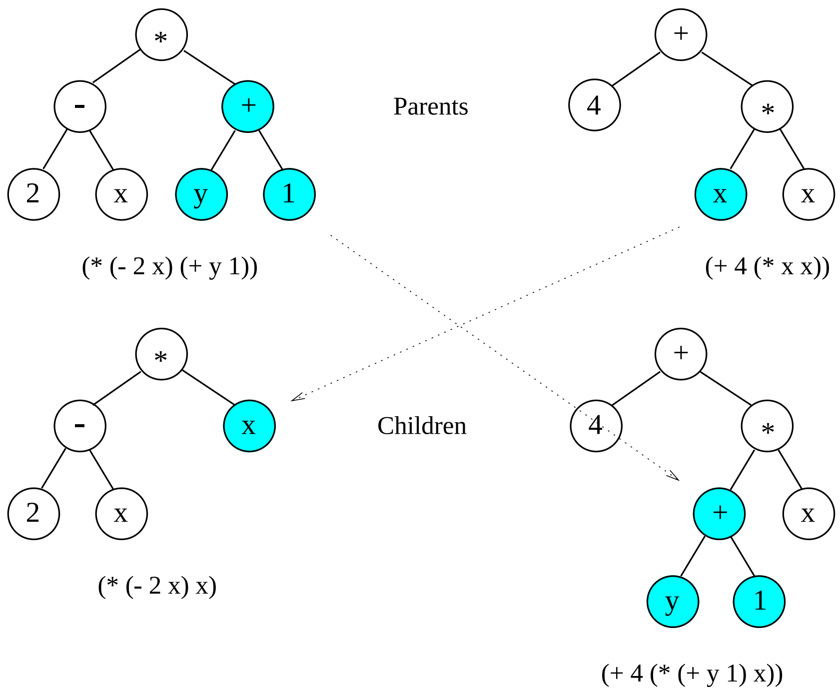

Figure 1: Crossover in tree-based GP. Subtrees in parents are selected and exchanged.

In recent years, the scope of genetic programming has expanded considerably and now includes evolution of linear and graph representations of programs as well, in addition to tree representations [3]. A strong motivation for investigating different program representations in GP is that for each representation form as well as for different learning methods in general there exist problem domains that are more suitable than others.

2.1 Linear Genetic Programming

In the experiments described below we use *linear* GP, a genetic programming approach with a linear representation of individuals. Its main characteristic in comparison to tree-based GP is that not expressions of a functional programming language (like LISP) but programs of an imperative language (like C) are evolved.

The use of linear bit sequences in GP again goes back to Cramer and his JB language [7]. Cramer later discarded his approach in favor of a tree representation. A more general linear approach was introduced by Banzhaf [2]. Nordin's idea of using machine code for evolution was the most radical "down-to-bones" approach [17] in this context. It was subsequently expanded [19] and led to the AIMGP (*Automatic Induction of Machine code by Genetic Programming*) system [3]. In AIMGP, individuals are manipulated directly as binary machine code in memory and executed directly without passing an interpreter during fitness calculation. This results in a significant speedup compared to interpreting systems. Due to their dependence on specific processor architectures, AIMGP systems are restricted presently in portability.

Our LGP system implements another variant of linear genetic programming. An individual program is represented as a variable-length string composed of simple C instructions. An excerpt of a linear genetic program is given below.

```
void ind(v)
  double v[8];
{
  ...

  v[0] = v[5] + 73;
  v[7] = v[0] - 59;       (I)
  if (v[1] > 0)
  if (v[5] > 21)
    v[4] = v[2] * v[1];
  v[2] = v[5] + v[4];     (I)
  v[6] = v[0] * 25;
  v[6] = v[4] - 4;
  v[1] = sin(v[6]);
  if (v[0] > v[1])        (I)
    v[3] = v[5] * v[5];   (I)
  v[7] = v[6] * 2;
  v[5] = v[7] + 115;      (I)
  if (v[1] <= v[6])
```

```
    v[1] = sin(v[7]);
}
```

The *instruction set* (or *function set*) of the system is composed of arithmetic operations, conditional branches and function calls. The general notation of each *instruction type* listed in Table 1 shows that — except for the branches — all instructions include implicitly an assignment to a variable v_i (*destination variable*). This makes the use of multiple program outputs easy in linear GP while in tree-based GP these side-effects have to be incorporated explicitly.

Instruction Type	General notation	
Arithmetic operation	$v_i := v_j\ op\ v_k \vert c$	$op \in \{+, -, *, /\}$
Conditional branch	$if\ (v_i\ cmp\ v_k \vert c)$	$cmp \in \{>, \leq\}$
Function call	$v_i := f(v_j)$	$f \in \{sin, cos, sqrt, exp, log\}$

Table 1: Instructions in LGP.

Instructions either operate on two variables (*operand variables*) or on one variable and one integer constant c. At the beginning of program execution these variables hold the program inputs and at the end the program output(s). Variables and constants together form the "terminal set" in linear GP. Each instruction is encoded into a four dimensional vector holding the instruction identifier, indices of all participating variables and a constant value (optionally), e.g. $v_i := v_j + c$ is represented as $(id(+), i, j, c)$. Since each vector component uses one byte of memory only the maximum number of variables is restricted to 256 and constants range from 0 to 255 at maximum. This representation allows an efficient recombination of the programs as well as an efficient interpretation.

Partially defined operations and functions are protected by returning a constant, artificially defined value (here 1) for all undefined inputs. Sequences of branches are interpreted as *nested branches* like in C which allows complex conditions to be evolved. If the condition of a branch or nested branch is false only *one* instruction is skipped, namely the next non-branch in the program. This treatment of conditionals has enough expressive power because leaving out or executing a single instruction can deactivate a lot of preceding effective code or reactivate preceding introns respectively (see Section 2.2).

The evolutionary algorithm of our GP system applies *tournament selection* and puts the lowest selection pressure on the individuals by allowing only *two* individuals to participate in a tournament. The loser of each tournament is replaced by a copy of the winner. In such a *steady state* EA the population size is always constant and determines the number of individuals created in one *generation*.

Figure 2 illustrates the *two-point string crossover* used in LGP for recombining two tournament winners. A segment of random position and random length is selected in each of the two parents and exchanged. If one of the resulting children would exceed the maximum length crossover is aborted and restarted with exchanging equally sized segments.

The crossover points only occur *between* instructions. *Inside* instructions the *mutation* operation randomly replaces either the instruction identifier, a variable or the constant (if existent) by equivalents from valid ranges. Constants are modified through a certain standard deviation (*mutation step size*) from the current value. Exchanging a variable, however, can have an enormous effect on the program flow which might be the reason why in linear GP high mutation rates have been experienced to produce better results.

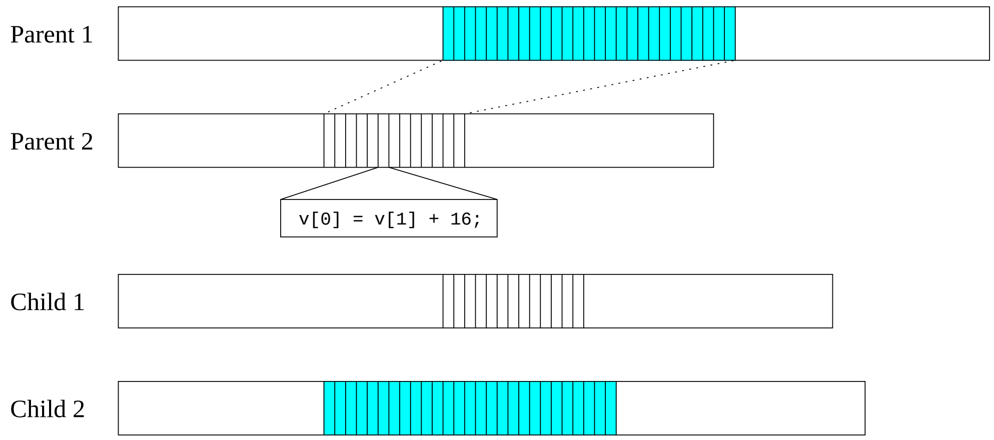

Figure 2: Crossover in linear GP. Continuous sequences of instructions are selected and exchanged between parents.

In genetic programming the maximum size of program is usually restricted to prevent programs from growing without bound. In our linear GP system the maximum number of instructions allowed per program has been set to 256. For all tested problems this configuration has been experienced to be a sufficient maximum length. Notwithstanding, individual programs of maximum length can still vary in size of their effective code (*effective length*, see Section 2.2). Since each instruction is encoded into four bytes of memory only one individual holds at most 1KB of memory. That makes the system quite memory efficient as well.

2.2 Removing Introns at Runtime

In nature *introns* denote DNA segments in genes with information that is not expressed in proteins. The existence of introns in eucaryotic genomes may be explained in different ways: (i) Since the information for one gene is often located on different *exons* (gene parts that *are* expressed) introns may help to reduce the number of destructive recombinations between chromosomes by simply reducing the probability that the recombination points will fall within an exon region [27]. In this way complete protein segments encoded by specific exons are more frequently mixed than interrupted during evolution. (ii) Perhaps even more important for understanding the evolution of higher organisms is the realization that new genetic material can be "tested" while retaining a copy of the original information in intron code.

After the DNA is copied the introns are removed from the resulting *messenger*-RNA that actually is participating in gene expression, i.e., protein biosynthesis. A biological reason for the removal of introns might be that the genes are more efficiently translated during protein biosynthesis in this way. Without being in conflict with ancient information held in introns, this might have an advantage, presumably through decoupling of DNA size from direct evolutionary pressure.

In analogy, an *intron* in a genetic program is defined as a program part without any influence on the calculation of the output(s) for *all* possible inputs. Other intron definitions common in genetic programming postulate this to be true only for the fitness cases [3, 18]. Introns in GP play a similar role as introns in nature in that they act as redundant code segments that protect advantageous building-blocks from being destroyed by crossover. Further, they also contribute to the preservation of the diversity of the population by retaining genetic material from direct evolutionary pressure.

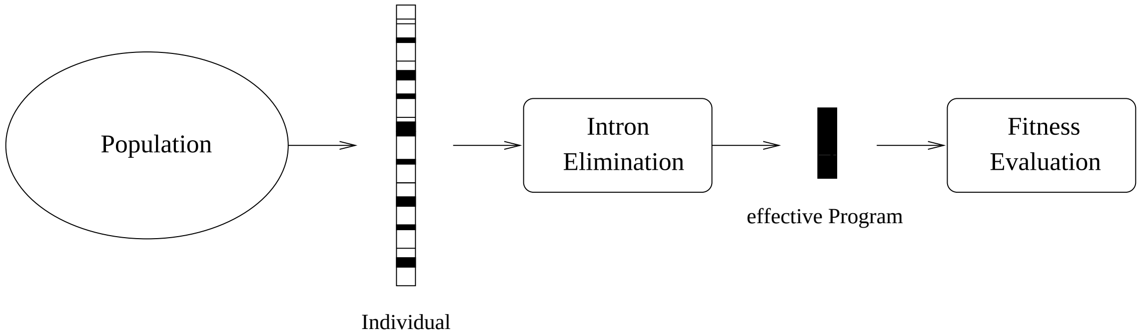

Figure 3: Elimination of intron code (white) in LGP. Only effective code (black) is copied to the execution buffer.

Two types of introns can be distinguished in linear GP. *Structural introns* denote single instructions that emerge from manipulating variables which are *not used* for the calculation of the outputs at that program position. In contrast to that, a *semantical intron* is an instruction or a sequence of instructions that manipulates relevant variables in that the state of the variables keeps the same as before. Three rather simple examples of semantical introns are:

(1)
```
v[0] = v[0] + 0;
```

(2)
```
v[0] = v[1] + 10;
v[0] = v[1] - 10;
```

(3)
```
if (v[0] > v[0])
v[i] = v[j] + c;
```

Example (3) is a special case in that the operation is *not executed* at all because the condition of the preceding branch is never fulfilled. Since it is much easier for the evolutionary algorithm to implement structural introns the rate of semantical introns in linear genetic programs can be expected to be comparatively low. In the following the term "introns" always denotes structural introns.

The program structure in linear GP allows introns to be detected and eliminated much easier than in tree-based GP. In LGP all structural introns are removed from a genetic program before evaluating the fitness cases. This is done by copying all *effective instructions*, i.e., non-introns, to a temporary program buffer. It does *not* affect the representation of the individuals in the population (see Figure 3). In this way the important property of the introns to protect the information holding code from being disrupted is preserved. In analogy to nature, the elimination of intron code allows linear genetic programs to be interpreted more efficiently.

The following algorithm detects all structural introns in a linear genetic program. Note that whether a branch is a structural intron or not only depends on the status of the operation that directly follows. In the example program from Section 2.1 all instructions marked with an `(I)` are introns provided that the program outputs are stored in `v[0]` and `v[1]` at the end.

Algorithm 2 (Intron detection):

1. Set V always contains all program variables that have an influence on the final program output at the

current position.

$V := \{ v_i \mid v_i \text{ is output variable } \}$

Start at the last program instruction and move backwards.

2. Mark the next operation with *destination* variable $v_i \in V$.

   If such an instruction is not found, $\rightarrow$ 5.

3. If the operation directly follows a branch or a sequence of branches mark these instructions, too, else remove v_i from V.

4. Insert the *operand* variables of new marked instructions in V if not already contained. $\rightarrow$ 2.

5. Stop. All *non-marked* instructions are introns.

All *marked* instructions are copied to form the *effective program*. The algorithm needs linear runtime $O(n)$ at worst where n is the maximum length of the genetic program. Actually, detecting and removing introns from a program only requires about the same time as calculating one fitness case. The more fitness cases are calculated the more this computational overhead will pay off.

By not executing the intron instructions during fitness evaluation a large amount of computation time is saved. A good estimate of the overall speedup in runtime is the factor

$$\alpha_{speedup} = \frac{1}{1 - p_{intron}} \tag{1}$$

where p_{intron} denotes the average intron percentage of a genetic program and $1 - p_{intron}$ the respective percentage of effective code. The percentage figure of all individuals in the population is computed by the algorithm as a side-effect and can be used for further statistical analysis.

Linear GP programs can be transformed into functional expressions by a successive replacement of variables starting with the last effective instruction as well. It is obvious that such a tree grows exponentially with the effective program length and can become extremely large in size. These trees normally contain many identical subtrees, too, since the more they grow the more instances of a variable are likely to be replaced by the next assignment. This might give an impression of the expressive power of the linear representation.

At least it demonstrates that there is a difference from evolving functional expressions like in traditional GP.

3 The Medical Datasets

In contrast to the neural network domain, performance results today are documented for a very selected number of problems in genetic programming. Many of these problems are artificial benchmarks where the optimal solution is known in advance and sufficient data examples are given. Often little or nothing is said about how results will hold in more realistic and noisy domains. Using real-world data instead guarantees results that are relevant for at least the tested problem domain. Once several problems from different domains have been tried the confidence is increased that the tested method is not depending on a specific problem.

In this contribution genetic programming is applied to six medical problems. Table 2 gives a brief description of the diagnosis problems and the diseases that are to be predicted. Medical diagnosis problems always describe classification tasks that are much more frequent in medicine than approximation problems.

Problem	Diagnosis task
cancer	benign or malignant breast tumor
diabetes	diabetes positive or negative
gene	intron-exon, exon-intron or no boundary in DNA sequence
heart	diameter of a heart vessel is reduced by more than 50% or not
horse	horse with a colic will die, survive or must be killed
thyroid	thyroid hyperfunction, hypofunction or normal function

Table 2: Medical diagnosis tasks of PROBEN1 benchmark datasets.

The datasets have been taken unchanged from an existing collection of real-world benchmark problems, PROBEN1 [20], that has been established originally for neural networks. The results obtained with one of the fastest learning algorithms for feed-forward neural networks (RPROP) accompany the PROBEN1 benchmark set to serve as a direct comparison with other methods. Comparability and reproducibility of the results is guaranteed by careful documentation of the experiments. Following the benchmarking idea and in order to increase the confidence in our results with linear GP the results for neural networks have been adopted here from [20] but have been verified exemplarily. The main objective is to realize a comparison as fair as possible between genetic programming and neural networks in medical data classification and in medical diagnosis respectively. We will show that for all problems discussed the performance of GP in generalization comes very close to or is even better than the results documented for neural networks in [20].

All PROBEN1 datasets originate from *UCI Machine Learning Repository* [5] and are organized as a sequence of independent example vectors divided into input and output values. To improve the comparability of results, the representation of the original (raw) datasets has been equalized in [20] in that values have been normalized, recoded and completed. All inputs are restricted to the continuous range [0,1] except for the gene dataset that holds -1 or 1 only. For the outputs a binary *1-of-m encoding* is used where each bit represents one of the m possible output classes of the problem definition. Only the value of the given output class takes 1 while the others are 0. It is characteristic for medical data that they almost always suffer from unknown attributes. For that reason in most of the UCI datasets missing inputs have to be artificially completed (by 0 in PROBEN1), e.g. 30% in case of the horse dataset.

Table 3 gives an overview of the specific complexity of each problem expressed in the number of problem attributes, continuous and discrete inputs, output classes and examples. Note that some attributes have been encoded into more than one input value.

Problem	Attributes	Inputs		Classes	Examples
		continuous	*discrete*		
cancer	9	9	0	2	699
diabetes	8	8	0	2	690
gene	60	0	120	3	3175
heart	13	6	29	2	303
horse	20	14	44	3	364
thyroid	21	6	15	3	7200

Table 3: Problem complexity of PROBEN1 medical datasets.

4 Experimental Setup

4.1 Genetic Programming

For each dataset and experiment 30 runs have been performed with LGP, all starting with the same configuration but with a different random seed. Table 4 lists the common parameter settings used for all problems.

For benchmarking, the partitioning of the datasets is adopted from PROBEN1. The *training set* always

Parameter	Setting
Population size	5000
Number of demes	10
Migration rate	5%
Classification error weight in fitness	1.0
Maximum number of generations	250
Crossover probability	90%
Mutation probability	90%
Maximum mutation step size for constants	± 5
Maximum program size	256 instructions
Initial maximum program size	25 instructions
Function set	$\{+, -, *, /, sin, exp, if >, if \leq\}$
Terminal set	$\{0, .., 255\} \cup \{v_0, .., v_{k-1}\}$ (k inputs)
Random seed	system time

Table 4: Parameter settings for LGP.

includes the first 50% of the examples from a dataset, the next 25% is defined as the *validation set* and the last 25% is the *test set*. In PROBEN1 three different compositions of each dataset are given concerning the order of the examples. This increases the confidence that results are not depending on a certain distribution of the data into training, validation and test set.

The fitness of an individual program is always evaluated over the complete training set. After each generation, the error of the best-so-far individual is calculated using the validation set in order to check its generalization ability *during* training. From these individuals the one with minimum validation error is tested on the test set once *after* the training is over.

Throughout this paper, *fitness F* of an individual program p is calculated as the *mean square error* (MSE) between the predicted output o_{ij}^{pred} and the desired output o_{ij}^{des} for all n training examples and m outputs:

$$F(p) = \frac{1}{n \cdot m} \sum_{i=1}^{n} \sum_{j=1}^{m} (o_{ij}^{pred} - o_{ij}^{des})^2 + \frac{w}{n} CE = MSE + w \cdot MCE \tag{2}$$

Additionally, a *classification error* (CE) is computed as the number of incorrectly classified examples. The mean classification error (MCE) is added to the fitness while its influence is controlled by a weight parameter w (see Equation 2 and Table 4). In this way, the classification performance of a program determines selection more directly while the MSE term still allows continuous fitness improvements.

To provide a fair comparison, the *winner-takes-all* classification method has been adopted from [20]. Each output class corresponds to exactly one program output. The class with the highest output value designates the response according to the 1-of-m output representation introduced in Section 3.

Since only classification problems are dealt with in this contribution the *test classification error* characterizing the generalization performance and the generation in which the individual with the minimum validation error appeared (*effective training time*) are of main interests.

4.1.1 Population Structure

In evolutionary algorithms the population of individual solutions may be subdivided into multiple subpopulations. Migration of individuals among the subpopulations causes evolution to occur in the population as a whole. Wright first described this mechanism as the *island model* in biology [28] and reasoned that in semi-isolated subpopulations, called *demes*, evolution progresses faster than in a single population of equal size. This inherent speedup of evolution by demes could be confirmed for EA [26] and GP in particular [25, 1]. One reason for this speedup may be that genetic diversity is better preserved in multiple demes between which migration is restricted. This has an influence on the probability that the evolutionary search gets stuck in a local minimum. A tendency towards a bad local minimum in one deme can be overcome by other demes with better search directions. A nearly linear increase in processing time can be achieved in evolutionary algorithms if demes are run in parallel on multi-processor architectures [1, 6].

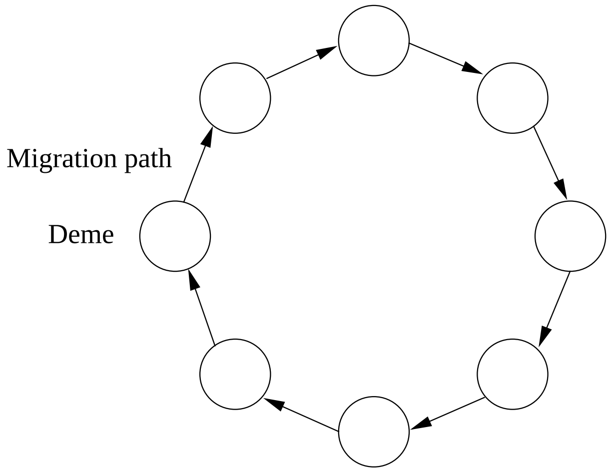

Figure 4: Stepping stone model of directed migration on a ring of demes.

A special form of the island model, the *stepping stone model* [13], assumes that migration of individuals is only possible between certain adjacent demes that are organized as graphs with *fixed* connecting links. Individuals can reach remote populations only after passing through these neighbors. In this way, the possibility that there will be an exchange of individuals between two demes depends on their distance in the graph topology. Common topologies are ring or matrix structures.

In our experiments the population is subdivided into 10 demes each holding 500 individuals. This partitioning has been experienced to be sufficient for investigating the effect of multiple demes. The demes are connected by a directed ring of migration links in that every deme has exactly one successor (see Figure 4). After each generation a certain percentage of best individuals from each deme, determined by the *migration rate*, emigrates into the successor deme by replacing the worst individuals there. By reproducing local best solutions into different demes of the population, learning may accelerate when these individuals are further developed simultaneously in different demes. The elitist migration strategy has the drawback to promote a premature loss of diversity caused by a faster proliferation of best individuals in the population. This is especially true, if migration between demes is not restricted to certain migration paths or occurs too frequently. Because of this trade-off between fast fitness progress and preservation of diversity, migration between demes has been organized in a ring topology here and modest migration rates of about 5 percent have been used.

4.2 Neural Networks

The experimental results in [20] have been achieved using standard multi-layer perceptrons (MLPs) with fully connected layers but without shortcut connections, e.g. without direct connections between the input and the output layer. Different numbers of hidden units and hidden layers (one or two) have been tried to find out the best net architecture for each dataset. The method applied for training was always RPROP learning [21], a backpropagation variant about as fast as Quickprop but with less ajustments of the parameters necessary. For the RPROP parameter settings and the special network architectures, see [20].

The generalization performance on the test set was computed for that state of the network which had minimum validation error during training. The number of *epochs*, i.e., the number of times the training examples are presented to the neural network, until this state denote the *effetive training time* of neural networks.

5 Results and Comparison

5.1 Intron Rate

Problem	Introns (%)		Effective Code (%)		Branches (%)		Speedup
	average	*stddev*	*average*	*stddev*	*average*	*stddev*	
cancer	65.45	2.79	34.55	2.79	23.19	0.85	2.9
diabetes	74.50	0.62	25.50	0.62	—	—	3.9
gene	90.49	1.07	9.51	1.07	21.74	0.73	10.5
heart	88.17	0.91	11.83	0.91	—	—	8.5
horse	90.82	0.42	9.18	0.42	—	—	10.9
thyroid	72.24	1.83	27.76	1.83	20.12	0.77	3.6

Table 5: Percentage of introns, effective code and branches per run with speedup factors for removing introns before program execution. Notable differences between problems.

As denoted in Section 2.2 the average intron rate of all individuals in the population has been calculated during each run. Table 5 documents the average share in intron code per run (in percent of the absolute program length) and the resulting speedup factor (using Equation 1) for the medical problems under consideration. In summary an intron rate of about **80%** has been observed which corresponds to an average decrease in *runtime* by the intron elimination of about factor **5**. This speedup is significant especially when operating with large datasets as they occur in medicine. Another benefit of intron elimination is that the higher *processing speed* of the genetic programs makes them very efficient in time-critical problem environments. Note again, that removing the introns as described in Section 2.2 can not have any influence on the fitness or classification performance of the programs.

It can also be seen from Table 5 that the respective share in the effective code strongly varies with the problem. In contrast to that, the differences in results between the three datasets tested for each problem were found to be only very small and are not specified here. Beyond that the standard deviation has proven to be amazingly small as well.

For some problems, including diabetes, heart, and horse, the best classification results (see below) have emerged without using the conditional branches. This might be due to the fact that if branches are not really necessary for a good solution then they promote rather specialized, i.e., less generalizing, solutions. The other problems, especially gene, have worked considerably better with branches. Except for the branches, all problems have been tried with the same function set and system configuration in general.

Concerning the execution time branches are very cheap instructions compared to operations and function calls. An additional amount of computation is left out with branches in that usually not all (conditional) operations of a program are executed at each training example. For problems solved with branches the average percentage of branches in a linear genetic program is given in Table 5. In general, the calculation of the relative speedup factors relies on the assumption that, on average, different instructions of the instruction set are homogeneously distributed in the population — including both intron code and effective code of the programs.

5.2 Generalization Performance

	GP						NN		
Problem	Validation CE (%)			Test CE (%)			Test CE (%)		Δ (%)
	best	*average*	*stddev*	*best*	***average***	*stddev*	***average***	*stddev*	
cancer1	1.71	2.45	0.34	0.57	2.18	0.59	1.38	0.49	-36.70
cancer2	0.57	1.39	0.40	4.02	5.72	0.66	4.77	0.94	-16.61
cancer3	1.71	2.62	0.45	3.45	4.93	0.65	3.70	0.52	-24.95
diabetes1	20.31	22.19	1.09	21.35	23.96	1.42	24.10	1.91	+0.58
diabetes2	21.35	23.21	1.33	25.00	27.85	1.49	26.42	2.26	-5.14
diabetes3	25.52	26.69	0.65	19.27	23.09	1.27	22.59	2.23	-2.17
gene1	7.81	11.16	2.30	9.21	12.97	2.24	16.67	3.75	+22.20
gene2	9.07	12.93	2.30	8.45	11.95	2.15	18.41	6.93	+35.09
gene3	7.18	10.77	2.11	10.09	13.84	2.09	21.82	7.53	+36.57
heart1	7.89	10.53	2.38	18.67	21.12	2.02	20.82	1.47	-1.42
heart2	14.47	18.58	2.39	1.33	7.31	3.31	5.13	1.63	-29.82
heart3	15.79	18.81	1.47	10.67	13.98	2.03	15.40	3.20	+9.22
horse1	28.57	32.40	2.22	23.08	30.55	2.24	29.19	2.62	-4.45
horse2	29.67	34.30	2.65	31.87	36.12	1.95	35.86	2.46	-0.72
horse3	27.47	32.65	1.94	31.87	35.44	1.77	34.16	2.32	-3.61
thyroid1	0.83	1.31	0.34	1.28	1.91	0.42	2.38	0.35	+19.75
thyroid2	1.11	1.62	0.31	1.44	2.31	0.39	1.91	0.24	-17.32
thyroid3	0.89	1.47	0.23	0.89	1.88	0.36	2.27	0.32	+17.18

Table 6: Classification error rates of GP and NN for PROBEN1 medical datasets. NN data taken from [20]. Difference Δ in percent from baseline NN data.

Table 6 shows the classification error rates obtained with genetic programming (GP) and neural networks (NN) respectively for the medical datasets discussed in Section 3. The best and the average classification error (CE) of all GP runs are documented on the validation and test set for each medical dataset, together with the standard deviation. Most interesting is a comparison with the test classification error of neural networks (reprinted from [20]). For that purpose the difference Δ between the average test errors of NN and GP is printed in percent, additionally. Unfortunately, the classification results on the validation set and the results of the best runs are not specified in [20].

Our results demonstrate that in general linear genetic programming is able to reach a quite similar generalization performance as was possible for multi-layer perceptrons using RPROP learning. The relatively small number of runs performed per dataset can, of course, only give an order of magnitude comparison. In addition, the results for GP might not be the absolutely best possible since the parameter settings — including the composition of the function set — have not been adjusted specifically for each problem. This has been done in order to show that even common adjustments of the GP algorithm can produce quite good results in general. In contrast to that the NN architecture applied in [20] has been adapted for each dataset in particular. Finally, the PROBEN1 datasets — especially the coding of the input and output values (see Section 3) — are prepared for being advantageous for neural networks but not necessarily for GP as well.

For the gene problem the test classification error (on average and standard deviation) has been found to be remarkably better with GP which is another indication that GP can handle very high input dimensions (given in Table 3) efficiently. In contrast to that, cancer emerged to be considerable more difficult for GP than for NN if the percentage difference in average test error is regarded.

The classification results for the three different datasets of each problem show that the difficulty of a problem may already change significantly with the distribution of the data into training, validation and test set. This is especially true for the test error that may differ much more with the data distribution than the validation error. For instance, the test error is much smaller for dataset heart2 than for heart1. For some datasets training, validation and test set also differ significantly in the representation of the data space. This may result in a strong difference between the validation error and the test error, as in case of cancer and heart.

5.3 Training Time

As stated above the *effective training time* specifies the number of (*effective*) generations or epochs respectively till the minimum validation error occured. One can see from Table 7 and Table 3 that the more complex a problem is or the more difficult it proves to be for GP or NN the longer is the effective training time. Even if a comparison between generations and epochs is rather difficult it is interesting to see that the effective training time for GP shows lower variation with respect to the different problems than the training

	GP		NN	
Problem	effective Generations		effective Epochs	
	average	*stddev*	*average*	*stddev*
cancer1	26	24	95	115
cancer2	26	25	44	28
cancer3	17	11	41	17
diabetes1	23	14	117	83
diabetes2	28	25	70	26
diabetes3	21	15	164	85
gene1	77	21	101	53
gene2	90	20	250	255
gene3	86	14	199	163
heart1	17	14	30	9
heart2	20	14	18	9
heart3	21	18	11	5
horse1	18	16	13	3
horse2	19	16	18	6
horse3	15	14	14	5
thyroid1	55	18	341	280
thyroid2	64	15	388	246
thyroid3	51	14	298	223

Table 7: Effective training time of GP and NN (rounded).

time for NN. Average and standard deviation have been found to be less variable with GP here.

A more important result is that in our GP experiments the effective training time could be reduced considerably by using demes (as described in Section 4), without leading to worse results in generalization during validation and testing. A comparable series of runs without demes but with the same population size has been performed for the first dataset of each problem. The average classification rates documented in Table 8 differ only slightly from the results obtained with a demetic population (see Table 6).

	GP without Demes					
Problem	Validation CE (%)			Test CE (%)		
	best	*average*	*stddev*	*best*	*average*	*stddev*
cancer1	1.14	2.05	0.50	1.15	2.85	1.22
diabetes1	19.27	21.43	0.72	20.31	24.39	1.75
gene1	7.68	11.03	2.99	8.95	12.56	3.09
heart1	7.89	11.01	3.00	18.67	22.27	2.94
horse1	26.37	32.40	1.92	21.98	30.73	3.51
thyroid1	0.72	1.27	0.43	1.22	1.96	0.54

Table 8: Classification error rates of GP without demes. Average results similar to results with demes (see Table 6).

Table 9 compares the effective training time using a panmictic (non-demetic) population with the respective results from Table 7 after the same maximum number of 250 generations. On average (and standard

	GP *with* Demes		GP *without* Demes		
Problem	effective Generations		effective Generations		Speedup
	average	*stddev*	*average*	*stddev*	
cancer1	26	24	62	67	2.4
diabetes1	23	14	62	53	2.7
gene1	77	21	207	42	2.7
heart1	17	14	68	75	4.0
horse1	18	16	59	63	3.3
thyroid1	55	18	200	36	3.6

Table 9: Effective training time of GP with and without demes. Significant speedup with demes.

deviation) the number of effective generations is reduced by using demes of about factor **3**. Thus, there is a faster convergence of the evolutionary algorithm to the state of optimal generalization performance. The main reason for this speedup may be in the elitist migration strategy applied here (*migration-of-the-best*) by which best solutions are reproduced and further developed simultaneously in more than one deme.

5.4 Further Comparison

Of course, reducing the (*relative*) training time on generation basis affects the *absolute* training time as well since runs may be stopped earlier. Comparing the absolute runtime in genetic programming and neural networks the fast learning algorithm applied for the feed-forward multi-layer perceptrons has been superior here. This is especially due to the large population size that has been used with GP in order to guarantee sufficient diversity as well as a sufficient number of demes. Moreover, since we concentrated on a comparison in optimal classification performance the configuration of our LGP system has not been optimized for runtime. If a small population size is used the intron elimination that accelerates LGP runs for several times (as shown above) may help to relax the difference in runtime between both techniques.

In contrast to neural networks, GP is not only capable of predicting outcomes but may provide insight into and a better understanding of the medical diagnosis by allowing an analysis of the resulting genetic programs [16]. Knowledge extraction from genetic programs is much more possible with programs that are compact in size and free from redundant information as resulting from the intron elimination in our linear GP system. In general, this property makes genetic programming a less opaque method than neural networks.

A further advantage of genetic programming is that in principle it does not require any form of specific architecture selection like multi-layer neural networks.

6 Discussion and Future Work

We have reported on linear genetic programming that has been applied successfully to a number of medical classification tasks. All datasets originate from a set of real-world benchmark problems established for neural networks in particular [20]. For genetic programming there is still a lack of a *standard set of benchmark problems*. Such a set would give the researchers the opportunity for a better comparability of their published methods and results. An appropriate benchmark set should be composed of *real-world* datasets taken from real problem domains as well as *artificial* problem definitions where the characteristics of the data are in most cases exactly known.

But a set of benchmark problems is not enough to guarantee comparability and reproducibility of results. A single parameter that is not published or an ambiguous description can make an experiment irreproducible and may let the results obtained with a method differ greatly. In many contributions either comparisons with other methods are not given at all or experiments with the methods compared to had to be reimplemented first. In order to make a direct comparison of published results easier a set of *benchmarking conventions* has to be defined, along with the benchmark problems. These conventions would describe standard ways of setting up and documenting an experiment, as well as measuring and documenting the results. A step in this direction has been taken by Prechelt for neural networks [20].

We have presented an efficient algorithm for the detection of introns in linear genetic programs. The elimination of introns before every fitness evaluation results in a significant decrease in runtime. In addition, the number of relevant generations of the evolutionary algorithm was reduced by using a demetic population together with an elitist migration strategy. Increasing the runtime performance of genetic programming with these techniques is especially important when operating with large datasets from real-world domains like medicine.

By using demes in genetic programming we experienced that the best generalization on the validation set is reached long before the final generation. Wasted training time can be saved if runs are stopped earlier. Appropriate *stopping rules* that monitor the progress in fitness and generalization over a period of generations are to be defined in this context.

The information about the effective size of the genetic programs can be used for *parsimony pressure.* In contrast to punishing the absolute size over the fitness function this does not work against the growth of intron code. Thus, introns may fulfill their function as a protection against destructive crossover operations while programs with short effective code are favoured by evolution. The basic idea behind parsimony pressure is that short solutions are expected to be more robust, i.e., generalize better on unknown data [15].

Acknowledgements

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

References

[1] D. Andre and J.R. Koza (1996) *Parallel Genetic Programming: A Scalable Implementation Using The Transputer Network Architecture.* In P.J. Angeline and K.E. Kinnear (eds.) *Advances in Genetic Programming 2*, 317–337 ,MIT Press, Cambridge, MA.

[2] W. Banzhaf (1993) *Genetic Programming for Pedestrians.* In S. Forrest (ed.) *Proceedings of the Fifth International Conference on Genetic Algorithms*, 628, Morgan Kaufmann, San Mateo, CA.

[3] 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.

[4] W.G. Baxt (1995) *Applications of Artificial Neural Networks to clinical medicine.* Lancet, 346:1135–1138.

[5] C. Blake, E. Keogh and C.J. Merz (1998) *UCI Repository of machine learning databases* [http://www.ics.uci.edu/~mlearn/MLRepository.html]. University of California, Department of Information and Computer Science, Irvine, CA.

[6] M. Brameier, F. Hoffmann, P. Nordin, W. Banzhaf and F.D. Francone (1999) *Parallel Machine Code Genetic Programming.* In W. Banzhaf, J. Daida, A.E. Eiben, M.H. Garzon, V. Honavar, M. Jakiela

and R.E. Smith *Proceedings of the International Conference on Genetic and Evolutionary Computation (GECCO99)*, Morgan Kaufmann, San Francisco, CA.

[7] N.L. Cramer (1985) *A representation for the adaptive generation of simple sequential programs.* In J. Grefenstette (ed.) *Proceedings of the First International Conference on Genetic Algorithms*, 183–187.

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

[9] R. Friedberg (1958) *A learning machine, part I. IBM Journal of Research and Development*, 2:2–13.

[10] R. Friedberg, B. Dunham and J. North (1959) *A learning machine, part II. IBM Journal of Research and Development*, 3:282–287.

[11] H.F. Gray, R.J. Maxwell, I. Martinez-Perez, C. Arus and S. Cerdan (1996) *Genetic Programming for Classification of Brain Tumours from Nuclear Magnetic Resonance Biopsy Spectra.* In J.R. Koza, D.E. Goldberg, David B. Fogel, and Rick L. Riolo (eds.) *Genetic Programming 1996: Proceedings of the First Annual Conference*, MIT Press, Cambridge, MA.

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

[13] M. Kimura (1953) *Stepping Stone Model of Population.* Annual Report of Nat. Gent. Japan.

[14] J.R. Koza (1989) *Hierarchical genetic algorithms operating on populations of computer programs.* In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, 768–774, Morgan Kaufmann, San Mateo, CA.

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

[16] P.S. Ngan, M.L. Wong, K.S. Leung and J.C.Y. Cheng (1998) *Using Grammar Based Genetic Programming for Data Mining of Medical Knowledge.* In J. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D.B. Fogel, M.H. Garzon, D.E. Goldberg, H. Iba and R.L. Riolo (eds.) *Genetic Programming 1998: Proceedings of the Third Annual Conference*, Morgan Kaufmann, San Francisco, CA.

[17] P. Nordin (1994) *A Compiling Genetic Programming System that Directly Manipulates the Machine-Code.* In K.E. Kinnear (ed.) *Advances in Genetic Programming*, 311–331, MIT Press, Cambridge, MA.

[18] P. Nordin, F. Francone and W. Banzhaf (1996) *Explicitly Defined Introns and Destructive Crossover in Genetic Programming.* In P. Angeline, K. Kinnear (eds.) *Advances in Genetic Programming II*, 111–134, MIT Press, Cambridge, MA,

[19] P.J. Nordin (1997) *Evolutionary Program Induction of Binary Machine Code and its Applications.* PhD thesis, University of Dortmund, Department of Computer Science.

[20] L. Prechelt (1994) PROBEN1 — *A Set of Neural Network Benchmark Problems and Benchmarking Rules.* Technical Report 21/94, University of Karlsruhe.

[21] M. Riedmiller and H. Braun (1993) *A Direct Adaptive Method for Faster Backpropagation Learning: the RPROP algorithm.* In *Proceedings of the IEEE International Conference on Neural Networks*, 586–591, San Francisco, CA.

[22] B.D. Ripley and R.M. Ripley (1997) *Neural Networks as Statistical Methods in Survival Analysis.* In R. Dybowski and V. Grant (eds.) *Artificial Neural Networks: Prospects for Medicine*, Landes Biosciences Publishers, Texas, USA.

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

[24] R.L. Somorjai, A.E. Nikulin, N. Pizzi, D. Jackson, G. Scarth, B. Dolenko, H. Gordon, P. Russell, C.L. Lean, L. Delbridge, C.E. Mountford and I.C.P. Smith (1995) *Computerized Consensus Diagnosis — A Classification Strategy for the Robust Analysis of MR Spectra. 1. Application to H-1 Spectra of Thyroid Neoplasma.* Magnetic Resonance in Medicine, 33:257–263.

[25] W.A. Tackett (1994) *Recombination, Selection and the Genetic Construction of computer programs.* Ph.D. thesis, University of Southern California, Department of Electrical Engineering Systems.

[26] R. Tanese (1989) *Distributed Genetic Algorithms.* In J.D. Schaffer (ed.) *Proceedings of the third International Conference on Genetic Algorithms*, 434–439, Morgan Kaufmann, San Mateo, CA.

[27] J.D. Watson, N.H. Hopkins, J.W. Roberts, J.A. Steitz and A.M. Weiner (1987) *Molecular Biology of the Gene.* Benjamin/Cummings Publishing Company, Inc.

[28] S. Wright (1943) Genetics, 28:114.