Adaptive Fitness Predictors in Coevolutionary Cartesian Genetic Programming

Michaela Drahosova idrahosova@fit.vutbr.cz
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Bozetechova 2, 612 66 Brno, Czech Republic

Lukas Sekanina sekanina@fit.vutbr.cz
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Bozetechova 2, 612 66 Brno, Czech Republic

Michal Wiglasz iwiglasz@fit.vutbr.cz
Brno University of Technology, Faculty of Information Technology, IT4Innovations Centre of Excellence, Bozetechova 2, 612 66 Brno, Czech Republic

https://doi.org/10.1162/evco_a_00229

Abstract

In genetic programming (GP), computer programs are often coevolved with training data subsets that are known as fitness predictors. In order to maximize performance of GP, it is important to find the most suitable parameters of coevolution, particularly the fitness predictor size. This is a very time-consuming process as the predictor size depends on a given application, and many experiments have to be performed to find its suitable size. A new method is proposed which enables us to automatically adapt the predictor and its size for a given problem and thus to reduce not only the time of evolution, but also the time needed to tune the evolutionary algorithm. The method was implemented in the context of Cartesian genetic programming and evaluated using five symbolic regression problems and three image filter design problems. In comparison with three different CGP implementations, the time required by CGP search was reduced while the quality of results remained unaffected.

Keywords

Cartesian genetic programming, coevolutionary algorithms, fitness prediction, symbolic regression, evolutionary design, image processing.

1 Introduction

The computation time an evolutionary design that is based on genetic programming (GP) requires for obtaining innovative results is enormous for complex real-world applications. One of the reasons is that the program's fitness is usually calculated over a large set of *fitness cases* (Koza, 1992). A fitness case corresponds to a representative situation in which the ability of a program to correctly produce the output value for a particular input can be evaluated. A fitness case consists of program inputs and target values expected from a perfect solution. The set of fitness cases (*training data*) is, however, only a small sample of the entire domain space. The choice of how many fitness

Manuscript received: 21 June 2016; revised: 14 March 2017, 4 September 2017, 11 May 2018; accepted: 14 May 2018.

cases (and which ones) to use is often a crucial decision because whether an evolved solution will generalize over the entire domain depends on this choice.

The time needed for evaluating a single fitness case depends on a particular application. Reducing the number of evaluated fitness cases (or fitness function calls) is thus crucial for the overall efficiency of the method.

Candidate programs can be coevolved with the so-called *fitness predictors* (predictors, for short) in order to reduce the evaluation time (Schmidt and Lipson, 2008). Predictors are small subsets of the training data and contain only selected fitness cases. During program evolution, predictors are coevolved to estimate the fitness of candidate programs (instead of applying the expensive objective fitness evaluation by means of complete training data). The benefits of this approach are numerous with respect to the standard GP: candidate programs are evaluated using fewer fitness cases, the number of candidate programs that have to be evaluated is reduced and the time required by GP search is shortened.

However, it is important to determine the most suitable parameters of coevolution, particularly the number of fitness cases that the fitness predictor contains (the predictor size, for short). This is a very time-consuming process as the predictor size depends on a given application and many experiments have to be performed to find its suitable size (Sikulova and Sekanina, 2012a, 2012b).

Our goal is to propose and evaluate a method capable of automated adaptation of the predictors (including their size) for a given problem and thus to reduce not only the time of evolution, but also the time needed to tune the evolutionary algorithm. We propose to adapt the predictor in the course of evolution on the basis of selected performance indicators, in particular, the evolution speed (see Section 3).

Although the proposed method is generally applicable in the context of GP, we will focus on its utilization in Cartesian genetic programming (CGP) becuase CGP is important for our target applications (see Section 4). CGP is a well-known form of GP developed by Miller and Thomson (2000). It uses a simple integer address-based genetic representation of programs in the form of a directed acyclic graph. CGP has been successfully applied to a number of challenging real-world problem domains (e.g., the design of electronic circuits, neural networks, and image operators). In a number of studies, CGP has been shown to be comparatively efficient in relation to other GP techniques (Miller and Turner, 2015).

We applied the proposed method in two types of problems: (i) symbolic regression and (ii) image filter design. Symbolic regression was included as it represents the standard approach for the performance comparison of GP systems. The case study (ii) was motivated by requirements on adaptive image filtering in embedded systems (see details in Section 4). The proposed method was compared with the standard CGP, coevolutionary CGP employing the constant-size predictor (Sikulova and Sekanina, 2012a, 2012b), and CGP with a randomly generated predictor. Our primary finding is that the coevolution of programs and adaptive fitness predictors leads to a reduction of the time required by the CGP search across all our test problems, while the quality of results remains unaffected.

In Section 2, previous relevant research is surveyed. The proposed method based on adaptive fitness predictors is presented in Section 3. Section 4 deals with the standard CGP and our motivation for evaluating the proposed approach in the context of CGP. Results of two case studies are presented in Section 5 (symbolic regression) and Section 6 (image filter design). Section 7 presents concluding remarks and the direction of our future research.

2 Related Work

This section briefly surveys concepts of fitness approximation, coevolution in evolutionary computing, and coevolution of fitness predictors.

2.1 Fitness Approximation

The goal of the GP software design and GP parameters tuning is to automatically obtain a program (solving the target problem with a predefined accuracy and robustness) in as short a time as possible. In practice, this time is measured as the number of evaluated fitness cases or fitness function calls. In order to reduce the computational complexity of expensive fitness evaluation, *fitness approximation* techniques have been developed.

One of them is *fitness modeling* which employs fitness models with different degrees of sophistication to reduce the evaluation time (Jin, 2005). A predefined model or coarse-grained simulation has been used to approximate the fitness value in the cases in which obtaining the objective fitness requires an expensive simulation or a physical experiment. Neural networks, support vector machines, decision trees and other machine learning methods can be used in order to efficiently approximate the objective fitness (Jin and Sendhoff, 2004). Sub-sampling of training data such as random subset selection (Gathercole and Ross, 1994), stochastic sampling (Nordin and Banzhaf, 1997), or dynamic topology-based selection (Lasarczyk et al., 2004) have also been studied in order to evaluate an individual on a smaller subset of fitness cases. However, it is not always clear when the benefits of fitness modeling can outweigh the cost (in particular, the overhead of model construction and solving a potential overfitting problem). The motivation for fitness modeling can be seen, not only in reducing the complexity of the fitness evaluation, but also in avoiding the explicit fitness definitions, coping with noisy data, smoothing the fitness landscape and promoting diversity (Schmidt and Lipson, 2008).

A closely related concept to fitness modeling is *fitness prediction*, which is a technique used to replace fitness evaluations by a lightweight approximation that adapts with the solution evolution. Fitness predictors cannot approximate the entire fitness landscape, but they are instead shifting their focus throughout the evolution. An algorithm that coevolves fitness predictors, optimized for the solution population, has been introduced for standard (tree-based) genetic programming in order to reduce the fitness evaluation cost and frequency; it is introduced by Schmidt and Lipson (2008).

2.2 Coevolutionary Algorithms

Coevolutionary algorithms (CoEAs) are characterized by comparing individuals on the basis of the outcomes of their interactions with other individuals (*subjective fitness* evaluation) instead of applying the objective fitness evaluation. Individuals can be evaluated by interacting with other individuals from the same population (*single population CoEA*). Or individuals in one population can be evaluated by interacting with individuals in one or several other populations (*multipopulation CoEA*). CoEAs are traditionally used to evolve interactive behavior which is difficult to evolve with an absolute fitness measurement. The state of the art of coevolutionary algorithms is summarized in Popovici et al. (2012).

Historically, the terms *cooperative* and *competitive* have been used to classify the domains in which coevolution is often applied. These terms appear from game theory, but they have not been appropriate for classifying problems over which CoEA operates nor for algorithms themselves. According to Popovici et al. (2012), problems are primarily divided into classes based on what constitutes a solution. Two types of problems are distinguished—*compositional* problems and *test-based* problems.

Coevolution applied to compositional problems sprang from the cooperative coevolutionary algorithms, wherein the originally stated aim was to attack the problem of evolving complicated objects by explicitly breaking them into parts, evolving parts separately and then assembling the parts into a working whole (Potter and De Jong, 2000). There are a number of successful applications of CoEAs to compositional problems. For example, in neuro-evolutionary algorithms, weights and structure of artificial neural networks are often coevolved (Stanley and Miikkulainen, 2004; Monroy et al., 2006; Shi and Wu, 2009; Shi, 2011).

Coevolution applied to compositional problems has also been proposed as a promising framework for solving high-dimensional optimization problems. However, it is not always clear how to decompose a problem into single variables. Yang et al. (2008) proposed a new problem decomposition strategy (the grouping based strategy), in order to better capture the variable interdependencies for complex nonseparable problems.

In coevolution applied to test-based problems, the quality of a potential solution is determined by its performance when interacting with a set of tests. Hillis (1990) introduced an approach that can automatically evolve subsets of fitness cases concurrently with a problem solution. Hillis employed a two-population coevolutionary algorithm in the task of designing a minimal sorting network. Subsets of fitness cases composed to evaluate candidate sorting networks were evolved simultaneously with the sorting networks. Evolved sorting networks were applied to evaluate the subsets of fitness cases. The fitness of each sorting network was measured by its ability to correctly solve fitness cases, while the fitness of the fitness cases subsets was better for those that could not be solved well by currently evolved sorting networks. It should be noted that the number of tests is typically assumed to be high in order to ensure an evolved solution will generalize over the entire domain.

The test-based problems are discussed in De Jong and Pollack (2004) and analyzed in connection with a multi-objective optimization in De Jong and Bucci (2008). Coevolving the fitness cases as the method of fitness modeling in GP has been studied in many application domains (Dolinsky et al., 2007; Gagné and Parizeau, 2007; Mendes et al., 2001) as well as in symbolic regression problems (Dolin et al., 2002; Pagie and Hogeweg, 1997; Schmidt and Lipson, 2006, 2008).

The minimal set of objectives that can provide satisfactory information about the structure of a problem is mentioned in connection with test-based problems in De Jong and Pollack (2004) and with compositional problems in Panait et al. (2006).

2.3 Coevolution of Fitness Predictors

The proposed method develops a specific form of coevolution introduced in Schmidt and Lipson (2006, 2008). Their method combines fitness prediction with coevolution to eliminate disadvantages of the classic fitness modeling, in particular the effort needed to train a fitness model and adapt the level of accuracy. In this coevolutionary algorithm, the objective fitness evaluation is replaced with a subjective fitness calculated by using a *fitness predictor*. Fitness predictors are coevolved in a second population in order to provide accurate fitness predictions. The population of solutions is evaluated with the current best fitness predictor while the population of fitness predictors evolves to minimize the difference between objective and subjective fitness when measured using the current population of solutions. This approach has been applied in the Eureqa software, which is successful in determining mathematical equations that describe sets of data in their simplest form (Schmidt and Lipson, 2009).

In Sikulova and Sekanina (2012b), coevolution of fitness predictors has been applied in order to accelerate the fitness evaluation in CGP. The fitness predictor encoding has been adopted in the form of a subset of the fitness cases set. Fitness predictors have been represented as a constant-size array of pointers to elements in the training data and operated using a simple genetic algorithm (GA). The coevolutionary algorithm has been adapted for CGP. Speedup of 2.0–5.4 has been reported in comparison with the standard CGP for five symbolic regression benchmarks and the results have been quite competitive with the tree-based GP (Schmidt and Lipson, 2008) in terms of the number of fitness evaluations needed to obtain a satisfactory solution.

The same coevolutionary CGP together with the *competitive coevolution* approach (Hillis, 1990) adapted for CGP have been used in the evolutionary design of salt-and-pepper noise suppressing filters (Sikulova and Sekanina, 2012a). Although the time of evolution was also reduced, a large number of experiments had to be accomplished in order to find the most advantageous size of the fitness predictor (the number of fitness cases in the predictor) for a particular task. An open problem is how to reduce this overhead.

In order to solve this problem, Sikulova et al. (2015) have introduced a new type of indirectly encoded fitness predictors which can automatically adapt the number of fitness cases necessary to evaluate the candidate programs. These fitness predictors have been represented in the form of functional expressions. The functional expression generates a certain number of indexes into the training data. Indexes then address specific fitness cases from the original training data which are selected for the prediction of the solution fitness. In order to evaluate the fitness of a candidate predictor, two features have to be observed: the prediction precision and the size of the predictor. The method has been evaluated using five symbolic regression benchmarks and compared with the original approach (Sikulova and Sekanina, 2012b). It was shown that the size of the predictor can automatically adapt to a particular benchmark. However, during evolution of fitness predictors, large fitness predictors can emerge and must be evaluated (and then refused for a larger size), and thus plenty of fitness case evaluations are wasted. Finally, Wiglasz and Drahosova (2016) have integrated the *phenotypic plasticity* principles into coevolution. The phenotypic plasticity is the ability of an individual to learn how to use its genotype in order to adapt to the environment (Baldwin, 1896). Inspired by this principle, a fitness predictor operated using a simple genetic algorithm employing the phenotypic plasticity has been introduced in order to adapt the number of fitness cases for candidate program evaluations.

3 Adaptive Fitness Predictors

This section introduces the concept of constant-size and adaptive-size fitness predictors in coevolutionary GP. It presents a new algorithm capable of dynamic adaptation of the predictor size, besides useful fitness cases selection, with respect to a particular problem.

The proposed algorithm employs four collections of individuals. There are two populations: 1) population of candidate programs and 2) population of fitness predictors. Furthermore, two archives are used: 1) archive of fitness trainers and 2) archive containing the current best evolved fitness predictor.

The archive of fitness trainers is used by the predictor population for the evaluation of evolved fitness predictors. It contains copies of selected candidate programs obtained

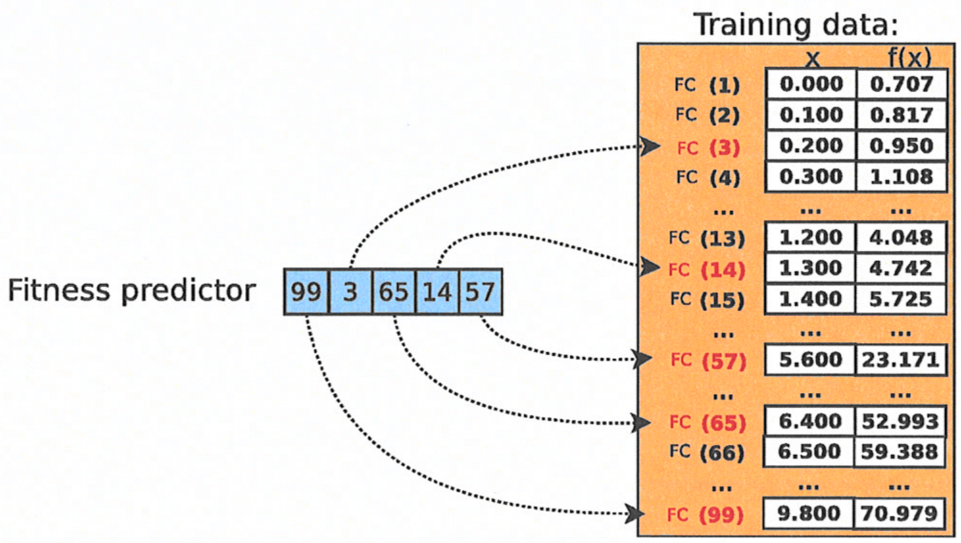


Figure 1: Example of fitness predictor addressing 5 fitness cases (FCs), where $k = 99$.

during the evolution. The fitness predictor from the other archive is used to evaluate the candidate programs.

3.1 Fitness Predictor

Let T be a training data set containing k fitness cases ($k = |T|$). Formally, fitness predictor P, $P \subseteq T$ is a small subset of the training data ($|P| \ll |T|$). It is encoded as an array of pointers to elements in the training data. Figure 1 illustrates how every pointer addresses one selected fitness case.

A good fitness predictor provides a fitness prediction which is fast and robust enough to differentiate any pair of candidate programs with high fidelity. The fitness value of the fitness predictor is (in our case) calculated using the mean absolute error of the objective fitness and subjective fitness of *fitness trainers*. Fitness trainers are selected copies of candidate programs that occurred during the program evolution.

Let us consider a symbolic regression problem where the candidate program fitness is represented as the number of hits. In common (noncoevolutionary) GP, the program fitness function (here the objective fitness is equal to the subjective fitness) is then defined as

$$f_{common}(s, T) = \sum_{j=1}^{k} g\left(y_j(s, T)\right), \text{ and} \tag{1}$$

$$g\left(y_j(s, T)\right) = 1 \text{ if } \left|y_j(s, T) - t_j\right| \leq \varepsilon; \text{ otherwise } 0, \tag{2}$$

where $y_j(s, T)$ is the response of candidate program s for the j-th fitness case from T, t_j is the target response, and ε, $\varepsilon \geq 0$ is a user-defined maximum error.

When the coevolutionary GP with fitness predictor P is employed, there are, in fact, two fitness functions for candidate program s. While the objective fitness function $f(s, T)$ uses the complete training data, the subjective fitness function $\hat{f}(s, P)$ employs only selected fitness cases. Formally,

$$f(s, T) = \frac{1}{k} \sum_{j=1}^{k} g\left(y_j(s, T)\right), \tag{3}$$

$$\hat{f}(s, P) = \frac{1}{m} \sum_{j=1}^{m} g\left(y_j(s, P)\right), \tag{4}$$

where m is the number of fitness cases in the fitness predictor P (i.e. m is the size of the predictor).

Let A be a set of programs in the trainer archive. The fitness value of predictor P is then expressed as

$$f_p(A, P) = \frac{1}{u} \sum_{i=1}^{u} \left| f(A_i, T) - \hat{f}(A_i, P) \right|, \tag{5}$$

where A_i is the i-th trainer and u is the number of programs (trainers) in the trainers' archive A.

3.1.1 Constant-Size Fitness Predictor

The basic framework supporting the coevolution of fitness predictors developed by Schmidt and Lipson (2008) introduces the fitness predictor encoding in the form of a constant-size array of pointers to elements in the training data. However, our previous work (Sikulova and Sekanina, 2012b, 2012a) revealed that the recommended predictor size differs from task to task. In order to find the most advantageous predictor size for a particular task, a large number of experiments have to be accomplished.

3.1.2 Adaptive-Size Fitness Predictor

In order to eliminate the problem of an expensive search for the most suitable predictor size for a particular task, we propose an approach which enables GA to dynamically modify the predictor size during coevolution in response to the program evolution progress.

Our approach employs a specific predictor encoding and predictor fitness evaluation. The predictor is encoded as a constant-size array of pointers to elements in the training data. The size of this array is equal to the total number of fitness cases. In order to obtain the fitness predictor with a particular size, pointers are read sequentially from the beginning and the reading stops after processing a given number of pointers specified by the *readLength* variable which thus defines the predictor size. If the pointer value is already included, it is skipped in order to prevent duplicating fitness cases in the predictor. The *readLength* variable is not part of the predictor encoding and is determined during the progress of the candidate program evolution.

The evolution of predictors is under the control of the simple genetic algorithm (section 5), where only the crossover operator is modified in such a way that the split point is always selected between the beginning of the encoded predictor and the position given by the *readLength* value.

In our approach, the *readLength* value is updated in two cases: (i) each time the best subjective fitness is improved in the program population; and (ii) after elapsing G_{update} generations without any update of *readLength* in the program population. The *readLength* value depends on the *phase of evolution* (see below) and its update consists of the four following steps:

1. Estimate the *phase of evolution*: we propose to characterize phases of evolution in terms of the evolution speed v which we define as follows:

$$v = \frac{\Delta f}{\Delta G}, \tag{6}$$

where Δf is the difference between objective fitness values of the current best program and the best program at the previous *readLength* update, and ΔG is the number of generations since the last *readLength* value update. It should be noted that as the evolution of programs is guided by the subjective fitness, its speed can be negative because it is calculated from the objective fitness. The phases of evolution are defined as follows:

- *improvement*: $v > 0$: the best-of-population objective fitness is increasing,
- *stagnation*: $v \approx 0$: the stagnation of the best-of-population objective fitness,
- *deterioration*: $v < 0$: the best-of-population objective fitness is decreasing.

2. Calculate the *inaccuracy of fitness prediction*: as the predictor can be based only on a few fitness cases (in the extreme case, on a single fitness case), this may cause the overfitting of predictors which renders the evolution unable to produce satisfactory programs. In order to prevent this behaviour, we monitor the prediction inaccuracy I which we define as the ratio between subjective and objective fitness:

$$I = \frac{\hat{f}}{f}. \tag{7}$$

3. Select the *update rule*: the rules are based on the following assumptions:
   (a) If the inaccuracy exceeds the threshold I_{thr} (i.e. $I > I_{thr}$), the predictor size is increased to prevent the overfitting.
   (b) If the *improvement* phase is detected, the predictor size is increased in order to make the prediction more accurate.
   (c) If the *stagnation* phase is detected, the predictor size is decreased. As *stagnation* can be caused by reaching a local optima, this rule increases the probability of leaving this point.
   (d) If the *deterioration* phase is detected, the predictor size is decreased. *Deterioration* is usually detected after the application of the rule (c) and it signals that the evolution is leaving the *stagnation* phase; the decreasing of the predictor size can accelerate this process.

   The purpose of the rules is to find a suitable predictor size, and at the same time, to ensure the progress of evolution. The parameters of the rules are specified in Sections 5 and 6.

4. Update the *readLength* value: the new *readLength* value is obtained by multiplying the previous value by coefficient C:

$$\mathit{readLength} \leftarrow \mathit{readLength} \cdot C, \tag{8}$$

   where C is dependent on the proposed rules as seen in Sections 5 and 6.

After the *readLength* value is updated, a new generation of predictors is constructed using the updated *readLength* value. It should be noted that the program objective fitness is evaluated only when calculating Formulas 6 and 7. The objective fitness is then computed for the program showing the best subjective fitness in the population.

3.2 Predictor Evaluation

The fitness predictors are evaluated using *fitness trainers* that are selected copies of candidate programs which occurred during the program evolution. The number of trainers in the archive is kept constant during the evolution. When initializing the coevolution,

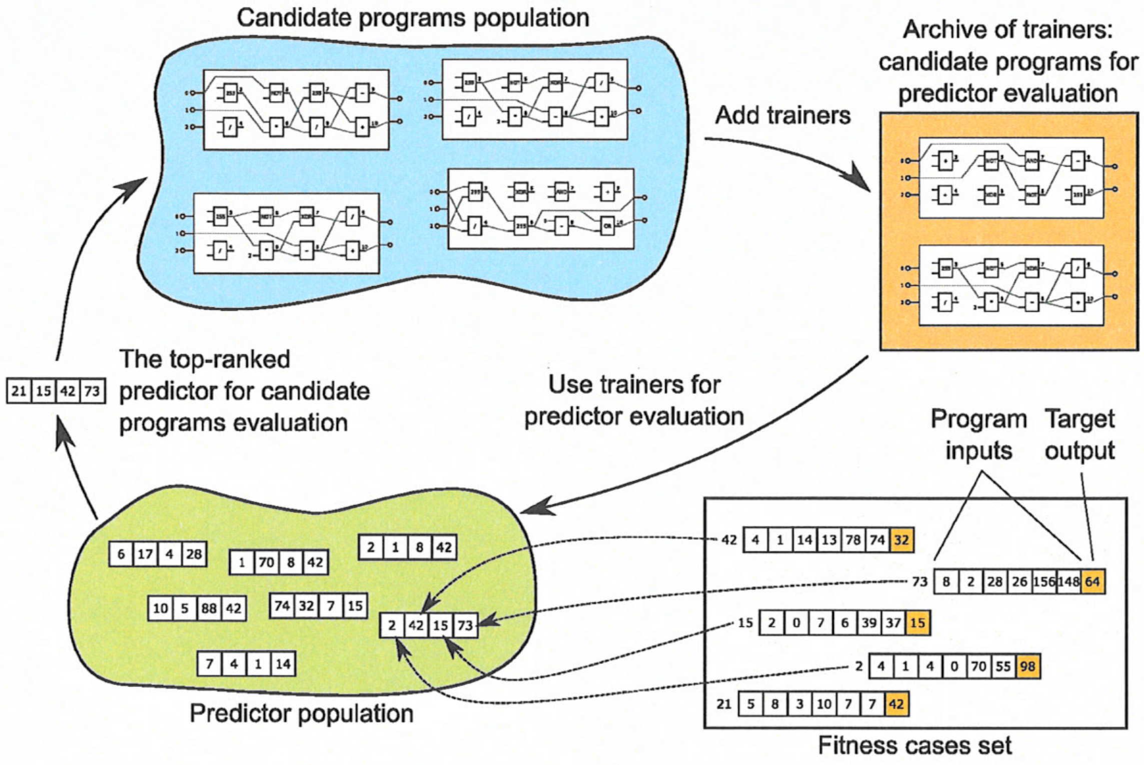


Figure 2: The overall coevolutionary scheme.

programs from the first generation are copied to the archive of trainers. If the archive of trainers is larger than the program population, the remaining trainers are generated randomly.

Trainers in the archive are updated periodically—the top-ranked candidate program is copied to the trainers archive if its subjective fitness value is better then the subjective fitness of the top-ranked trainer. This new trainer replaces the oldest one in the trainers archive and its objective fitness is evaluated. This approach leads to maintaining a representative sample of the program population (due to the copies of top-ranked candidate programs) as well as increasing the diversity of programs in the archive of trainers (due to the randomly generated trainers).

3.3 Coevolutionary Algorithm

Figure 2 shows the overall interaction scheme and Algorithm 1 provides a pseudocode of key procedures of the proposed coevolutionary algorithm.

Candidate programs are stored in *ProgPop* population. The goal of the program evolution is to find a program minimizing the difference between its responses and desired responses. Candidate programs are evaluated using predictors that are coevolved in the second population (*PredPop*). The predictors are trained using a set of *fitness trainers A* containing selected copies of candidate programs and randomly generated programs.

In the *CoevolutionInitialization* procedure, *ProgPop*, *PredProg*, and *A* are randomly initialized. Predictors from *PredPop* and trainers from *A* are evaluated and the top ranked predictor (*TRP*) is determined. After these initialization steps, procedures *ProgramEvolution* and *PredictorEvaluation* are started. Their synchronization will be described in Sections 5 and 6.

Algorithm 1: Pseudocode for coevolution of the population of programs and the population of predictors

```
procedure CoevolutionInitialization
    ProgPop ← random initialization                    // Program population initialization
    PredPop ← random initialization                    // Predictor population initialization
    $A \leftarrow ProgPop \cup$ random initialization   // Trainers archive initialization
    for $t \in A$ do                                   // Evaluate trainers
        $f(t, T)$
    end
    for $P \in PredPop$ do                             // Evaluate predictor population
        $f_P(A, P)$
    end
    TRP ← SelectTopRanked(PredPop)                     // TRP – Top Ranked Predictor
    $BRP_0 \leftarrow SelectTopRanked(A)$              // BRP – Best Ranked Program
    $R \leftarrow BRP_0$                               // R – Best known solution
end
procedure UpdateReadLength
    $v = \frac{f(BRP_G,TRP) - f(BRP(G_{last\_update}),TRP)}{G - G_{last\_update}}$
    $I = \frac{\hat{f}(BRP_G,TRP)}{f(BRP_G,T)}$
    $C \leftarrow SelectCoefficient(v, I)$
    readLength ← readLength · C
    Reconstruct top-ranked-predictor using new readLength
    $G_{last\_update} \leftarrow G$
end
procedure ProgramEvolution
    for $G = 1$ to $G_{max}$ do
        for $s \in ProgPop$ do                         // Evaluate program population
            $\hat{f}(s, TRP)$
        end
        $BRP_G \leftarrow SelectTopRanked(ProgPop)$
        if $\hat{f}(BRP_G, TRP) > \hat{f}(BRP_{G-1}, TRP)$ then
            Insert $BRP_G$ to $A$
            if $f(BRP_G, T) > f(R, T)$ then
                $R \leftarrow BRP_G$
            end
            call UpdateReadLength
        else if no update in last $G_{update}$ generations then
            call UpdateReadLength
        end
        ProgPop ← create new population
    end
    set TerminatingFlag
    return $R$
end
procedure PredictorEvolution
    repeat
        for $P \in PredPop$ do                         // Evaluate predictor population
            $f_P(A, P)$
        end
        TRP ← SelectTopRanked(PredPop)
        PredPop ← create new population
    until TerminatingFlag is set
end
program Coevolution
    call CoevolutionInitialization
    call ProgramEvolution and PredictorEvolution
end
```

The *ProgramEvolution* procedure starts by evaluating all candidate programs using subjective fitness function $\hat{f}$ which employs *TRP*. The subjective fitness values of the current best ranked program (*BRP*) and previous *BRP* are then compared, and if an improvement in subjective fitness is detected, the current *BRP* is inserted into trainers' archive A. This program is also compared against the best-known solution (R) in terms of the objective fitness. The best-known solution (R) is then the result of the whole coevolution. If conditions discussed in Section 3.1.2 are met, the *UpdateReadLength* procedure is called to modify the *readLength* variable. The pseudocode of this procedure formalizes the detailed description given in Section 3.1.2. Finally, a new population of candidate programs is generated. For example, in our case studies, only mutation is employed as it is the standard genetic operator in CGP. The user parameter G_{max} specifies the number of repetitions for all the aforementioned steps.

In the *PredictorEvaluation* procedure, the predictor population is evaluated using trainers from A followed by updating and storing the top ranked predictor. Then, the new generation of fitness predictors is created by means of a single-point crossover and mutation, and the evolution loop continues with the next iteration until the terminating flag is set on.

4 Cartesian Genetic Programming and Motivation for Case Studies

In order to evaluate the proposed method, we selected CGP because (i) CGP is relevant for our application-oriented research in the area of evolvable hardware and (ii) implementations of standard, as well as coevolutionary CGP, are available for a fair comparison. This section introduces the principles of CGP and the case studies used for experimental evaluation.

4.1 Cartesian Genetic Programming

Cartesian genetic programming has been introduced and developed by Miller and Thomson (2000). Compared to standard tree-based GP, CGP uses a simple integer-based representation of programs that are treated as directed acyclic graphs. This representation is especially useful for evolving digital circuits as it naturally captures the circuit physical structure and supports multiple outputs, subgraph sharing and various types of elementary circuit components. Although CGP requires only a simple and an easy to implement search algorithm, it is highly flexible for many applications and its performance is comparable to other forms of GP (Miller and Turner, 2015).

In standard CGP, candidate programs are modeled in a matrix of $n_c \times n_r$ programmable elements (nodes). Each node is programmed to perform one of n_a-input functions defined in the set Γ. The number of primary inputs, n_i, and outputs, n_o, of the program is defined for a particular task. Each node input can be connected either to the output of a node placed in previous l columns or to one of the program inputs. Feedback is not allowed in standard CGP. The search is usually performed using a simple $(1 + \lambda)$ evolutionary algorithm, where λ is usually between 1 and 20 (Miller and Thomson, 2000). Every new population consists of (i) the best individual of the previous population and (ii) its λ offspring created by using a mutation operator which modifies up to h genes of the chromosome. An example of a Cartesian program and its encoding in the task of image filter design is given in Figure 6.

4.2 Test Problems

Symbolic regression is a common problem used to evaluate the performance of GP-based systems. In addition to symbolic regression, the proposed method is evaluated in the task of automated image filter design, emerging in the context of adaptive embedded systems.

Hardware implementations of CGP are typically developed with the aim of either (i) enabling autonomous system adaptation at the hardware level, or (ii) reducing the execution time in comparison with a pure software implementation (Salvador et al., 2013; Dobai, 2014; Sekanina et al., 2011). Dobai (2014) showed that when properly accelerated in a field programmable gate array, CGP can evolve a unique image filter for every frame of a video played with a resolution of 427 × 240 pixels. In this task, CGP can generate and evaluate over 9,200 candidate filters per second, and each of them is evaluated using a 128 × 128 pixel image taken from the previous frame. As most of the time is spent on evaluating candidate filters, the evaluation must be highly optimized at the hardware level by means of multiple pipelined fitness units. The search algorithm is implemented as a program for an embedded microprocessor. In order to enable the adaptive video filtering for higher image resolutions and frame rates, it is important to improve the search algorithm which is currently the standard mutation-based CGP. There is no reason to accelerate the search operators in the hardware as the performance bottleneck is in the candidate filter evaluation. However, if the number of pixels in the training image or the number of candidate filters requiring the evaluation could be reduced by a smart search strategy, the time of evolution would be shortened.

The standard CGP has been successful in evolutionary image filter design. However, it is crucial to reduce the computational overhead of the method. It is proposed to coevolve the fitness predictors (representing small subsets of pixels of the training image) with Cartesian programs (representing desired image filters). In order to avoid a computationally expensive search for the most suitable setup of coevolutionary CGP, the proposed method will adapt the predictor size in the course of coevolution.

5 Case Study 1: Symbolic Regression

Five symbolic regression benchmark functions (F1–F5, taken from Sikulova and Sekanina, 2012b) were selected as training data sources for the evaluation of the proposed method:

$$\begin{aligned}
&\mathrm{F1}: f(x) = x^2 - x^3, && x = [-10:0.1:10]\\
&\mathrm{F2}: f(x) = e^{|x|}\sin(x), && x = [-10:0.1:10]\\
&\mathrm{F3}: f(x) = x^2 e^{\sin(x)} + x + \sin\left(\tfrac{\pi}{x^3}\right), && x = [-10:0.1:10]\\
&\mathrm{F4}: f(x) = e^{-x}x^3\sin(x)\cos(x)(\sin^2(x)\cos(x) - 1), && x = [0:0.05:10]\\
&\mathrm{F5}: f(x) = \tfrac{10}{(x-3)^2+5}, && x = [-2:0.05:8]\,.
\end{aligned}$$

In order to form the training data, 200 equidistant distributed samples were taken from each function. The user-defined acceptable errors ε are as follows: F1, F2: 0.5; F3: 1.5; F4, F5: 0.025. The requested number of hits is at least 96%.

5.1 Experimental Setup

The proposed coevolution scheme employing adaptive-size predictors (ASP) is compared with the original constant-size predictors (CSP), CGP with a random constant predictor (RP), and standard CGP ($\mathrm{CGP_{STD}}$) without coevolution.

The setup of the program evolution in all CGP versions is used according to the literature (Sikulova and Sekanina, 2012b), that is, $\lambda = 12$, $n_i = 1$, $n_o = 1$, $n_c = 32$, $n_r = 1$, $l = 32$, every node has two inputs (i_1, i_2), $\Gamma = \{i_1 + i_2, i_1 - i_2, i_1 \cdot i_2, \frac{i_1}{i_2}, \sin(i_1), \cos(i_1), e^{i_1}, \log(i_1)\}$, and $h = 8$.

Table 1: Rules used to adapt the *readLength* parameter in symbolic regression.

Priority	Condition	Coefficient C
1.	$I > I_{thr}$; $I_{thr} = 2.7$	1.20
2.	$\lvert v \rvert \leq 0.001$	0.90
3.	$v < 0$	0.96
4.	$0 < v \leq 0.1$	1.07
5.	$v > 0.1$	1.00

The fitness function for program evolution with the standard CGP (i.e., no coevolution) is represented as the number of hits (Formula 1). In coevolutionary CGP, the objective fitness (f) and subjective fitness ($\hat{f}$) are computed using Formulas 3 and 4.

In CGP with CSP, GA performing the predictor evolution employs 12 fitness cases in chromosome and 32 individuals in the predictor population. It uses a 2-tournament selection, a single-point crossover and a mutation probability 0.2. The same setup is used for coevolution with ASP, except specifying the number of fitness cases, which is variable. CGP with RP uses 12 fitness cases for prediction.

Our previous experiments have revealed that a frequent interaction (generation to generation) between populations does not lead to programs with the desired quality in a reasonable time because of very fast changes in involved populations. One generation in the predictor evolution that executed each 100 generations of the program evolution was enough to produce satisfactory predictors (Sikulova and Sekanina, 2012b).

In ASP, the predictor size is adapted as follows. The *readLength* value is initialized with 5 genes, its minimum size is limited to 5, and the maximum is the total number of fitness cases. The *readLength* value is updated every time when the subjective fitness is improved, or after $G_{update} = 5000$ generations since the last update.

Decision conditions and their priority for updating *readLength* are given in Table 1. Coefficients used in the decision procedure (Table 1) were determined experimentally using several short runs of the algorithm. They are identical in both case studies except in the condition considering I_{thr}. This setup allows us to reflect different ranges of the fitness scores in our case studies.

All experiments were performed on a cluster equipped with Intel Xeon E5-2680v3 (2.5-GHz) CPUs. Evolution of programs and evolution of predictors are executed in independent threads. In both case studies, the CPU time (user time) reported in tables and figures is the *sum* of runtimes of both threads which corresponds to the total time required by a coevolutionary search.

5.2 Results

The algorithms are compared in terms of the success rate (the number of runs that gives a solution with a predefined quality), the number of generations and the number of fitness case evaluations to converge (in order to compare the computational cost) and CPU time. Table 2 gives the median values calculated from 100 independent runs for each benchmark F1–F5.

Table 2 shows that ASP has the highest success rate for all benchmarks. The computational cost, represented as the CPU time necessary to find a solution, is the lowest in the case of benchmarks F1 and F2 and is comparable to CSP in benchmarks F3–F5. It should be noted that CGP with CSP requires performing many experiments to find

Table 2: Median values out of 100 runs for standard CGP (CGP_{STD}), coevolutionary CGP with CSP, CGP with ASP, and CGP with RP. For each benchmark, the best result is marked in bold font.

	Algorithm	F1	F2	F3	F4	F5
Success rate	CGP_{STD}	100%	100%	91%	5%	27%
	CSP	100%	100%	100%	33%	43%
	ASP	100%	100%	100%	94%	100%
	RP	100%	100%	69%	0%	3%
Generations to converge (median)	CGP_{STD}	$8.66 \cdot 10^3$	$3.09 \cdot 10^4$	$1.17 \cdot 10^5$	$\mathbf{4.13 \cdot 10^6}$	$3.25 \cdot 10^6$
	CSP	$\mathbf{2.08 \cdot 10^3}$	$\mathbf{1.07 \cdot 10^4}$	$2.60 \cdot 10^4$	$1.13 \cdot 10^7$	$7.32 \cdot 10^6$
	ASP	$2.47 \cdot 10^3$	$1.13 \cdot 10^4$	$\mathbf{2.50 \cdot 10^4}$	$5.60 \cdot 10^6$	$\mathbf{2.68 \cdot 10^6}$
	RP	$2.74 \cdot 10^3$	$2.00 \cdot 10^4$	$2.13 \cdot 10^5$	—	$4.46 \cdot 10^6$
CPU time (seconds) to converge (median)	CGP_{STD}	8.01	$2.85 \cdot 10^1$	$1.08 \cdot 10^2$	$6.46 \cdot 10^3$	$4.76 \cdot 10^3$
	CSP	$2.07 \cdot 10^{-1}$	1.18	**2.90**	$\mathbf{1.21 \cdot 10^3}$	$9.69 \cdot 10^2$
	ASP	$\mathbf{1.46 \cdot 10^{-1}}$	$\mathbf{7.22 \cdot 10^{-1}}$	3.25	$4.03 \cdot 10^3$	$1.19 \cdot 10^3$
	RP	$1.55 \cdot 10^{-1}$	2.00	$2.26 \cdot 10^1$	—	$\mathbf{4.44 \cdot 10^2}$

the most advantageous size of the predictor, while CGP with ASP can adjust the size of the predictor during each single run in response to a particular task.

Although CGP with RP does not spend resources on the evolution of the predictor population, it requires more generations to converge, and in the end consumes more CPU time than the coevolutionary approaches. Moreover, its overall ability to find a satisfactory solution is worse than standard CGP and thus it is suitable only for simple tasks. In the case of benchmark F4, it was not able to converge at all, and only 3 out of 100 runs discovered a solution for benchmark F5.

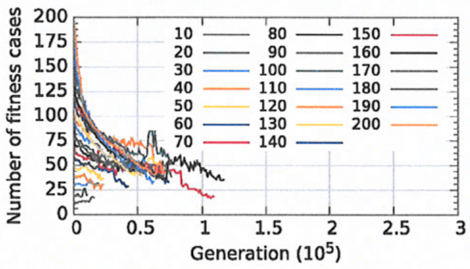
Finally, all coevolutionary approaches outperform the standard CGP in terms of CPU time required to converge and thus accelerate the design process.

5.3 Ability to Adapt the Number of Fitness Cases

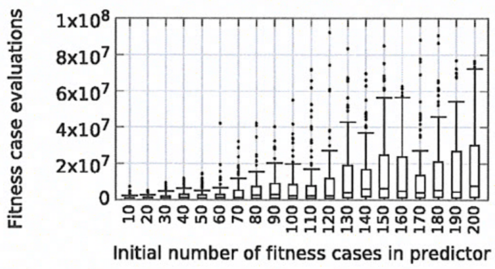
In order to confirm that the proposed algorithm is able to adapt the predictor size for a given task, we plot the progress of the average number (out of 100 independent runs) of fitness cases in the top-ranked predictor during the evolution flow with respect to the initial predictor sizes. Figure 3 shows that the size converges independently of the initial predictor size and the final predictor size differs for each benchmark.

The success rate is almost identical for every initial predictor size setting. In the case of benchmarks F1–F3, a larger initial predictor size leads to more fitness case evaluations required to find an acceptable solution (see Figure 3). This does not hold for benchmarks F4 and F5, where all settings lead to a comparable number of evaluations. The reason is that the predictor size converges in approximately 10^5 generations, while it takes more time (approx. $3.7 \cdot 10^6$ generations) to find a satisfactory solution (Table 2), so the effect of a different predictor size at the beginning of the evolution is negligible. It should be noted that a satisfactory solution for benchmark F1 was obtained in fewer generations than was necessary for the predictor size to converge.

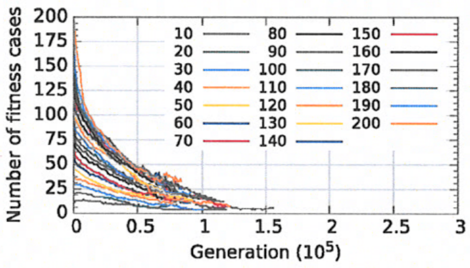
In general, it is advantageous to begin with a lower number of fitness cases in the predictor, which, in some cases, leads to a lower number of evaluations and thus the design process acceleration. On the other hand, if the initial predictor size is too small



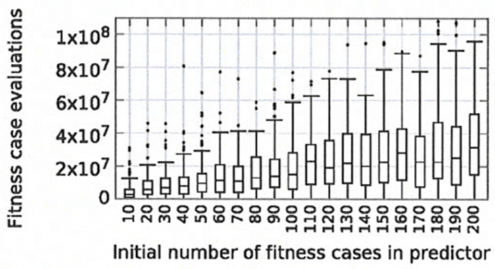
(a) Benchmark F1.



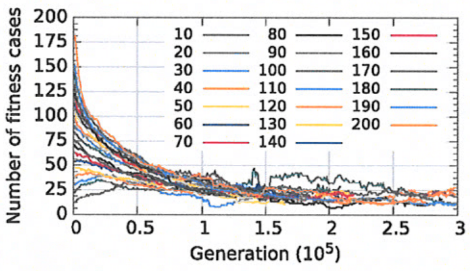
(b) Benchmark F1.



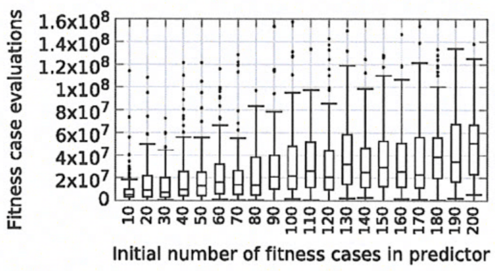
(c) Benchmark F2.



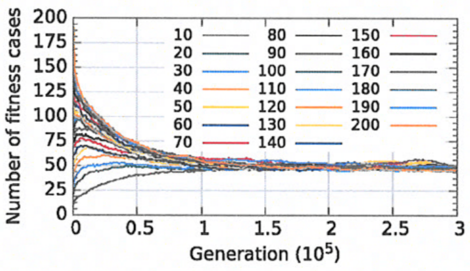
(d) Benchmark F2.



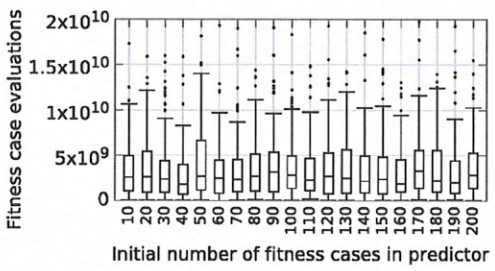
(e) Benchmark F3.



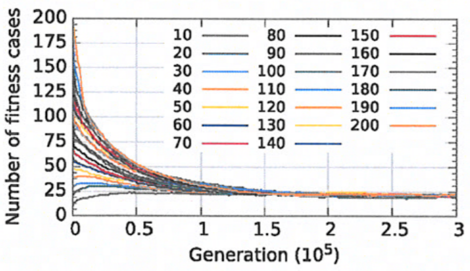
(f) Benchmark F3.



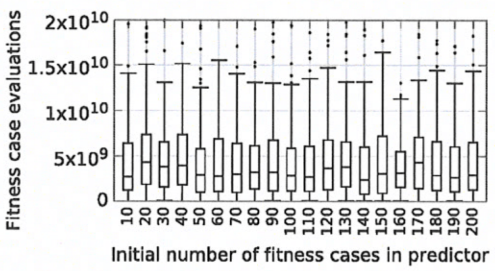
(g) Benchmark F4.



(h) Benchmark F4.

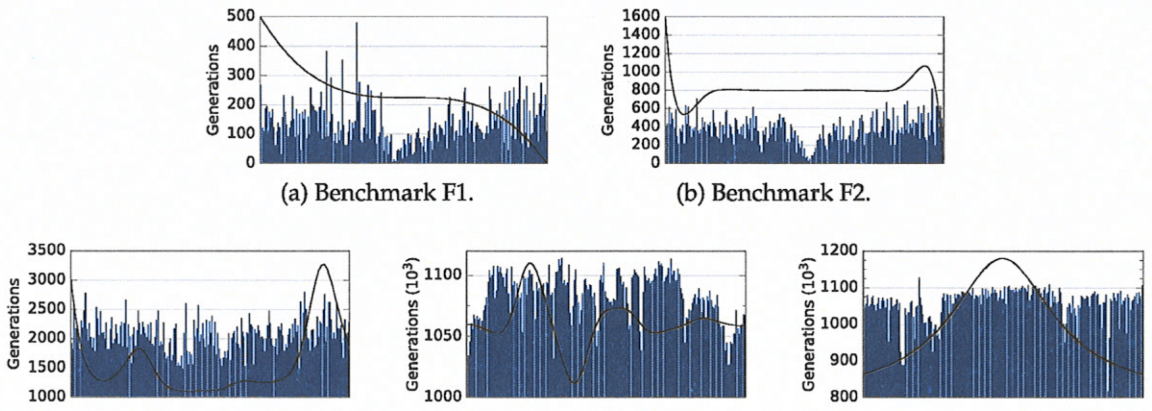


(i) Benchmark F5.



(j) Benchmark F5.

Figure 3: Different initial predictor sizes in the task of symbolic regression: the average number of fitness cases in predictors and the number of fitness case evaluations necessary to find an acceptable solution.



(a) Benchmark F1. (b) Benchmark F2.

(c) Benchmark F3. (d) Benchmark F4. (e) Benchmark F5.

Figure 4: Frequency (y-axis) of all fitness cases (x-axis) in predictors used for evaluation of programs. The black curves are benchmark functions F1–F5.

to find an acceptable solution, it will be automatically increased without a significant impact on the run time.

5.4 Predictor Behavior

Figure 4 shows the frequency of fitness cases addressed by the top-ranked predictors during the coevolution flow out of 100 independent runs for each symbolic regression task. For benchmarks F1 and F2, predictors focus more on peaks and valleys than on flexes. On the other hand, in the cases of F3–F5, the samples are well distributed over the data set. One should now consider that all fitness cases addressed by the predictor will be focused only on the interesting regions (peaks and valleys) of the training data. Such a predictor would lead to a very high error. It should be noted that this characteristic is desirable in Hillis's competitive coevolutionary approach (Hillis, 1990), but is improper when one requires obtaining very close subjective and objective fitness values. Furthermore, the fitness cases addressed by the fitness predictors are variable in response to the program evolution flow. The program evolution forces the predictors to contain easy as well as difficult fitness cases for a particular program.

6 Case Study 2: Image Filter Design

Evolutionary design of low-level image filters by means of CGP resulted in novel designs showing very good quality of processing and low implementation cost in comparison with conventional image filters (Sekanina et al., 2011). Various approaches have been proposed to accelerate the image filter evolution (Salvador et al., 2013; Dobai, 2014). We will show that a further reduction of the CGP runtime is possible by introducing fitness prediction into a coevolutionary design framework.

The proposed approach is evaluated in the design of edge detectors and impulse noise filters. We distinguish two basic types of impulse noise. Pixels corrupted by a *salt-and-pepper noise* can take only the minimum or maximum values from a given range. For images corrupted by *random-valued shot noise*, the noisy pixels have an arbitrary value. These two types of noise are illustrated in Figures 5b and 5c. In addition to basic noise suppressing filters, CGP is able to evolve advanced filters (Sekanina et al., 2011), for example, *dilate*, *erode*, *motion*, *unsharp* filters, or an *edge detector* (see Fig. 5d).

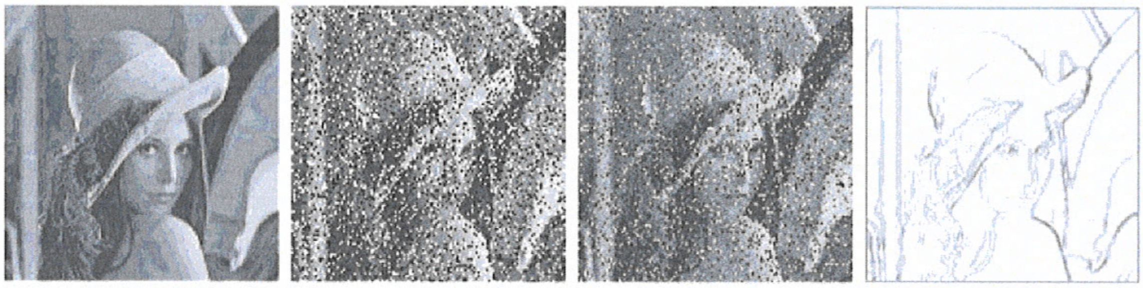


Figure 5: Examples of salt-and-pepper noise (S&P), random-valued shot noise (RVS), and edge detection (by Sobel operator).

Table 3: List of node functions for image filter evolution.

Code	Function	Description	Code	Function	Description
0	255	constant	8	$i_1 \gg 1$	right shift by 1
1	i_1	identity	9	$i_1 \gg 2$	right shift by 2
2	$255 - i_1$	inversion	A	$swap\,(i_1, i_2)$	swap nibbles
3	$i_1 \vee i_2$	bitwise OR	B	$i_1 + i_2$	+ (addition)
4	$\overline{i_1} \vee i_2$	bitwise $\overline{i_1}$ORi_2	C	$i_1 +^S i_2$	+ with saturation
5	$i_1 \wedge i_2$	bitwise AND	D	$(i_1 + i_2) \gg 1$	average
6	$\overline{i_1 \wedge i_2}$	bitwise NAND	E	$max\,(i_1, i_2)$	maximum
7	$i_1 \oplus i_2$	bitwise XOR	F	$min\,(i_1, i_2)$	minimum

This section presents the coevolutionary approach to image filter design, experimental setup, and results that are compared with other CGP-based implementations.

6.1 Cartesian Genetic Programming for Image Filter Design

In the case of the image filter design using CGP, as summarized in Sekanina et al. (2011), candidate filters operate over a filtering window consisting of 3×3 pixels. Each candidate filter can use up to nine 8-bit inputs (i.e., $n_i = 9$). It produces a single 8-bit pixel (i.e., $n_o = 1$). Table 3 gives a set of functions working over two pixels i_1 and i_2 that are used for image filter evolution in the literature. Figure 6 shows an example of a candidate filter and its encoding in CGP.

In the fitness function, the goal is to maximize the peak signal-to-noise ratio (PSNR) between the uncorrupted version of the training image (a golden image) and the result of filtering by candidate filter s. The PSNR is defined as:

$$\mathrm{PSNR}(s, T) = 10 \cdot log_{10} \frac{255^2}{\frac{1}{k} \sum_{(i,j) \in T} (v_{i,j}(s) - w_{i,j})^2}, \tag{9}$$

where k is the number of filtered pixels, v denotes the image produced by candidate filter s and w denotes the golden image. The training set T thus consists of k fitness cases (all pixels in the image), where one fitness case involves the filtering window of 3×3 pixels from the training image and a corresponding center pixel from the golden image, which represents the desired output of the image filter.

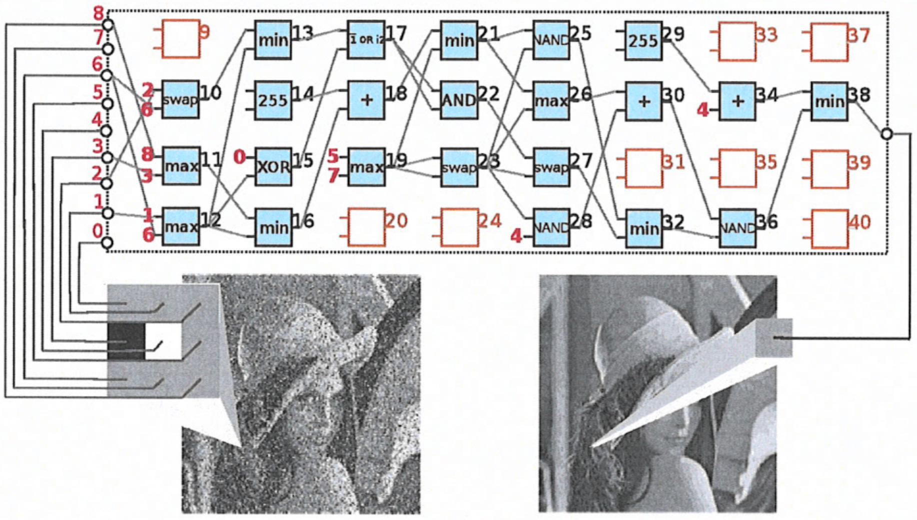


Figure 6: Candidate filter in CGP, where $l = 1, n_c = 8, n_r = 4, n_i = 9, n_o = 1, n_a = 2, \Gamma$ is according to Table 3.

In the coevolutionary algorithm, we replace the objective fitness function (i.e., PSNR) with a fitness prediction. The subjective fitness (PSNR_{sub}) is then defined as:

$$\text{PSNR}_{\text{sub}}(s, P) = 10 \cdot log_{10} \frac{255^2}{\frac{1}{m}\sum_{(i,j)\in P}(v_{i,j}(s) - w_{i,j})^2}, \tag{10}$$

where $m, m \leq k$ is the number of fitness cases (filtering windows with corresponding center pixels from the golden image) in predictor P.

6.2 Experimental Setup: CGP

CGP is used according to the literature (Sekanina et al., 2011), that is, $n_c = 8$, $n_r = 4$, $l = 1, n_i = 9, n_o = 1, n_a = 2, \lambda = 7, h = 5$ and Γ contains the functions from Table 3. The main objective is to evaluate the impact of parameters of coevolution on the quality of results. As the impact of CGP parameters has been analyzed in the literature, it is not our aim to deal with it here.

In order to evaluate the proposed approach, three types of image filters are evolved using CGP: 1) salt-and-pepper noise filters, 2) random-valued shot noise filters, and 3) edge detectors. In the case of the salt-and-pepper and random-valued noise filter design, 16 noise intensities are considered, that is, the Lena training image with resolution 256 × 256 pixels was corrupted by 5% up to 80% (step value 5%) noise to obtain the desired training images. Once the evolution is completed, evolved filters are tested using 12 different test images (Gonzalez et al., 2009) containing the same type of noise.

6.3 Experimental Setup: Coevolution

The coevolution setup is used according to our previous experiments in the image filter design (Sikulova and Sekanina, 2012a). Our solution exploits the assumption that by using a proper predictor evolution setup, the interaction between populations (each of them running in its own thread in our implementation) will occur at a desired time. Our experiments with different settings of the predictor evolution have revealed that 16 fitness trainers (in the archive) and 32 fitness predictors (in the predictor population) are enough to produce satisfactory predictors. This setup produces approximately one

Table 4: Rules used to adapt the *readLength* parameter in image filter design.

Priority	Condition	Coefficient C
1.	$I > I_{thr}; I_{thr} = 1.2$	2.00
2.	$\lvert v \rvert \leq 0.001$	0.90
3.	$v < 0$	0.96
4.	$0 < v \leq 0.1$	1.07
5.	$v > 0.1$	1.00

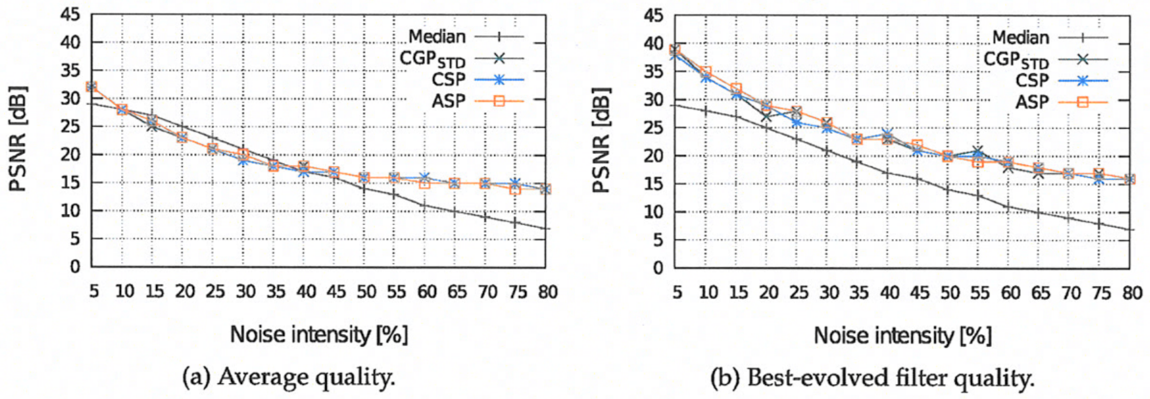


Figure 7: Quality of filtering (salt-and-pepper noise) using 12 test images.

generation in the predictor evolution after 64 generations of the program evolution. Both populations are thus naturally and indirectly synchronized.

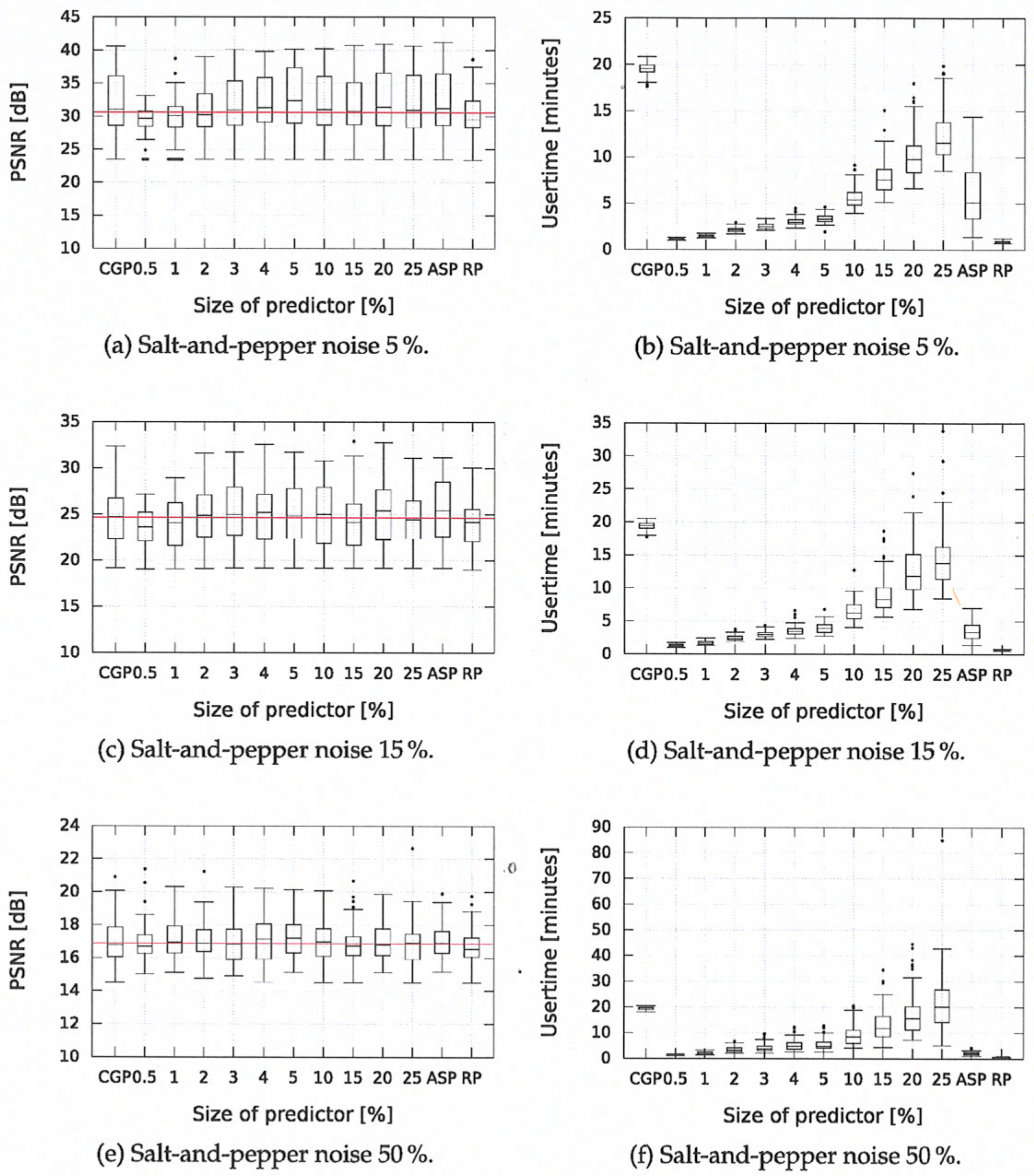
The evolution of fitness predictors is conducted using a simple GA. A new generation of predictors consists of 8 top-ranked predictors from the previous generation, 8 randomly generated predictors, and 16 offspring (created using a 2-tournament selection and a single-point crossover).

For CGP with CSP a proper number of fitness cases have been investigated and discussed for a particular task (see Section 6.5). In the case of ASP, the *readLength* value is initialized with 3% of all fitness cases. The role of the initial value is discussed in Section 6.6. Decision conditions for establishing the new *readLength* value are summarized in Table 4. The remaining setup was reused from Case Study 1.

All considered algorithms are compared in terms of the filtering quality of evolved filters and the execution time. Statistics are calculated from 100 independent runs for each approach and setup. The experiments were performed on a cluster equipped with Intel Xeon E5-2665 (2.4-GHz) CPUs.

6.4 Ability to Find a Satisfactory Solution

Firstly, we compare four approaches to the image filter design in terms of the quality of filtering. Figure 7 shows the average and best PSNR values for 12 test images obtained by filters evolved (in $3 \cdot 10^4$ generations) using standard CGP (CGP_{STD}), CGP with CSP (10% pixels), CGP with ASP, and a conventionally designed median filter. Coevolutionary approaches provide filters with a comparable quality of filtering with the standard CGP. Furthermore, all CGP approaches are able to provide filters with better PSNR than the conventional median filter.



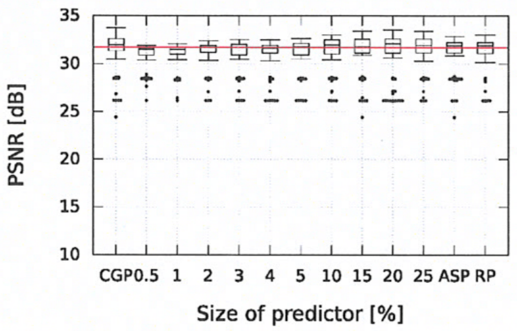
(a) Salt-and-pepper noise 5 %.
(b) Salt-and-pepper noise 5 %.
(c) Salt-and-pepper noise 15 %.
(d) Salt-and-pepper noise 15 %.
(e) Salt-and-pepper noise 50 %.
(f) Salt-and-pepper noise 50 %.

Figure 8: Salt-and-pepper noise: the quality of filtering (PSNR) and the time of evolution for filters evolved using the standard CGP, CGP with CSP, CGP with ASP, and CGP with RP—the boxplots created from 100 runs with $3 \cdot 10^4$ generations each.

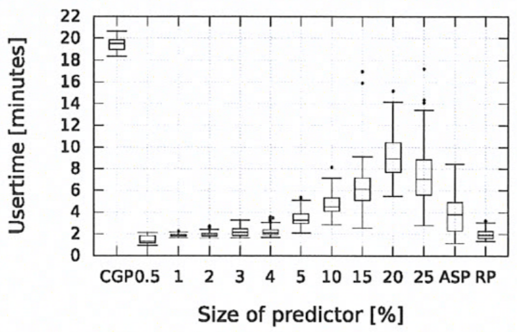
6.5 Constant-Size and Adaptive-Size Predictor

This section deals with a proper predictor size selection for a particular task in terms of balancing the ability to find the filter with satisfactory quality and evolutionary design cost. Figures 8 and 9 compare the quality of filters (PSNR) and the total CPU time (the user time) spent by the considered approaches.

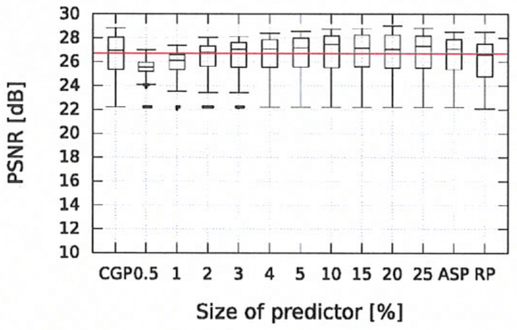
Figure 8 summarizes the results obtained for salt-and-pepper noise filters (noise intensity 5%, 15%, and 50%). For example, in the case of 5% noise intensity, the quality of filtering provided by the standard CGP, CGP with CSP (using 5% pixels), and CGP with ASP, are almost identical. CGP with CSP is 6.4 times faster than the standard CGP while



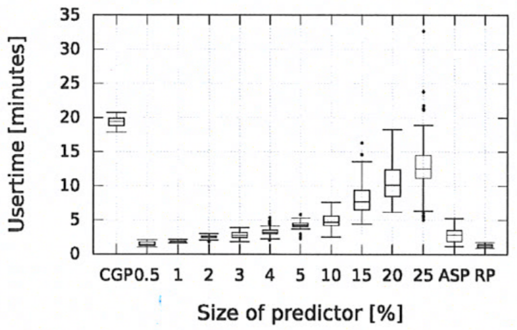
(a) Random-valued shot noise 5 %.



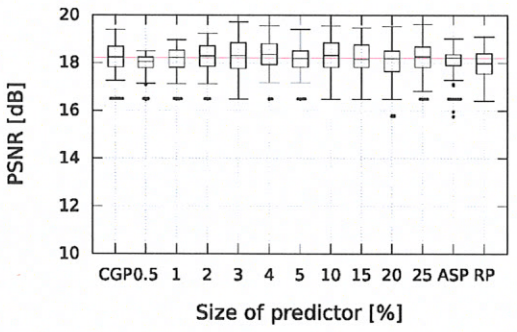
(b) Random-valued shot noise 5 %.



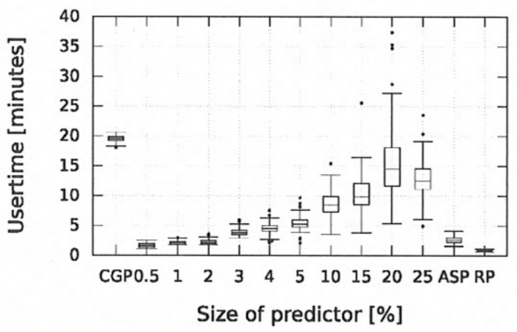
(c) Random-valued shot noise 15 %.



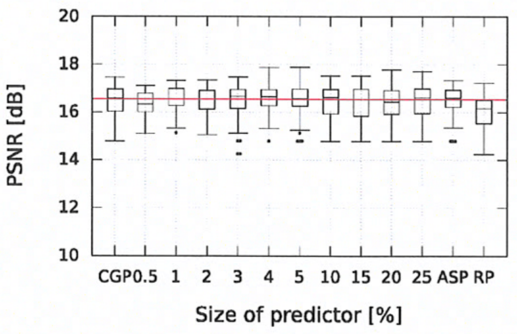
(d) Random-valued shot noise 15 %.



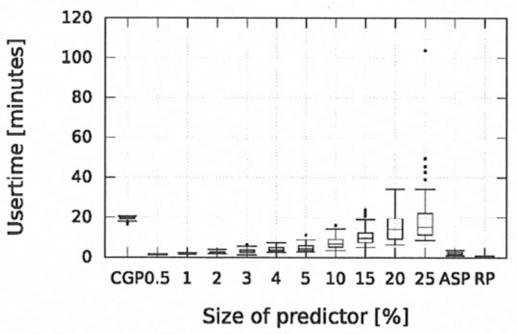
(e) Random-valued shot noise 50 %.



(f) Random-valued shot noise 50 %.



(g) The edge detector.



(h) The edge detector.

Figure 9: Random-valued shot noise and edge detectors: the quality of filtering (PSNR) and the time of evolution for filters evolved using the standard CGP, CGP with CSP, CGP with ASP, and CGP with RP—the boxplots created from 100 runs with $3 \cdot 10^4$ generations each.

CGP with ASP is only 2.4 times faster. In the case of 15% noise intensity, the proper size of the predictor is only 2% of all fitness cases; and CGP with this setting of the predictor is 8.5 times faster than the standard CGP. CGP with ASP is 5.4 times faster. Finally, for 50% noise intensity, CGP with CSP (1% pixels) and CGP with ASP provide a very similar speedup.

Figure 9 shows the same type of results, but only for random-valued shot noise (with intensities 5%, 15%, and 50%) and edge detectors. For example, in the case of 5% noise intensity, the proper predictor size is 20% of all fitness cases and CGP employing this predictor size is 2.1 times faster than the standard CGP. CGP with ASP accelerates the evolution 4.5 times. In the case of edge detection, CGP with CSP (using 3% of all fitness cases) provides the same quality as the standard CGP, while the time of evolution is reduced 6.4 times. CGP with ASP is 10.6 times faster than the standard CGP.

CGP with RP is the fastest method if a given number of generations have to be produced. However, the obtained filters provide lower quality compared to the coevolutionary approaches with the same predictor size.

It can be concluded that the standard CGP shows the worst performance under the conditions of our experiments. In the case of coevolutionary CGP, the proper size of the predictor differs from task to task. Many independent runs of CGP with a different setting of CSP have to be performed in order to find the most suitable predictor size for a particular task. CGP with ASP eliminates the need to perform these multiple experiments.

6.6 Examining Behaviour of Adaptive-Size Predictor

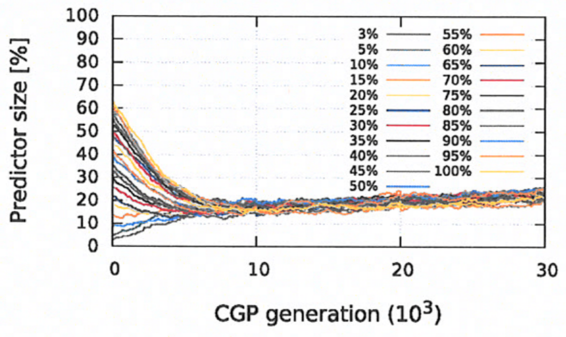
In order to confirm that the proposed approach is able to adapt the predictor size to a given task, we plot the progress of the average number (out of 100 independent runs) of fitness cases in the top-ranked predictor during the course of evolution with respect to the initial predictor size. It can be seen in Figure 10 that the average size of the predictor converges to a specific value independent of its initial size. Furthermore, the converged average predictor size differs for each benchmark.

In general, it is advantageous to begin with a lower number of fitness cases in the predictor, which in some cases leads to a lower number of fitness cases evaluations, thus shortening the design time. On the other hand, if the initial size is too small to find an acceptable solution, it will be automatically increased without a significant impact on the run time.

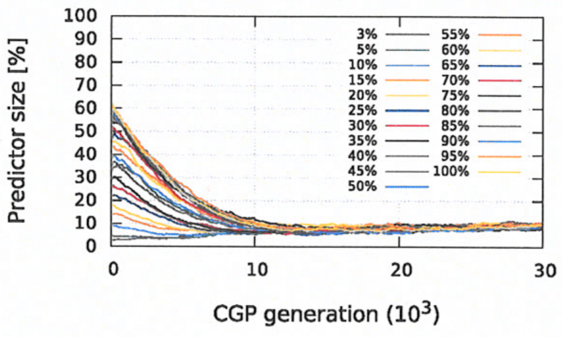
The number of fitness cases addressed by predictors is developing in response to the course of program evolution. Figure 11 shows the progress of the best (objective and subjective) fitness values during a typical run of the 15% salt-and-pepper filter design using CGP with CSP and ASP. It can be seen that the coevolutionary algorithms produce dynamic changes of the objective fitness while evolving a solution. Aside from the program evolution progress, the changes of the best-of-population subjective fitness value are caused by updating the current best-evolved fitness predictor. Moreover, the best-evolved program may not be present in the last population—the best-evolved program is always preserved as a part of the trainers' archive.

As the average size of the adaptive-size predictor converges to a specific value for each particular task, the actual size of the predictor in a single run is developing in reaction to the program evolution progress (Fig. 11b).

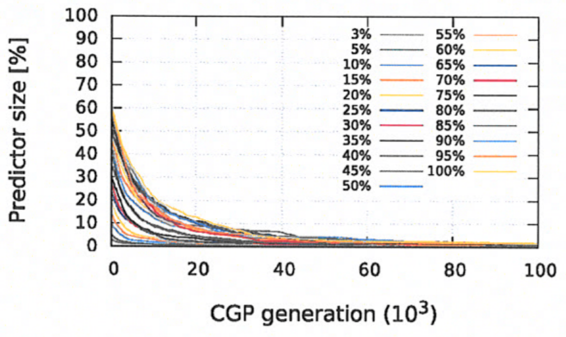
In order to understand the behaviour of predictor data samples, we plot a 2D bar graph (see Figure 12) portraying the frequency of fitness cases addressed by predictors, which were used during the course of evolution for program fitness prediction (100



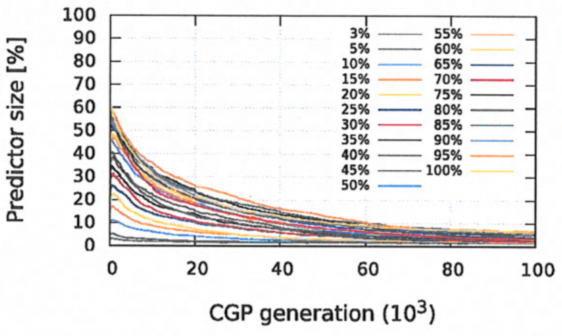
(a) Salt-and-pepper noise 5 %.



(b) Salt-and-pepper noise 15 %.



(c) Salt-and-pepper noise 50 %.



(d) The edge detector.

Figure 10: Convergence curves for various initial predictor sizes averaged from 100 independent runs.

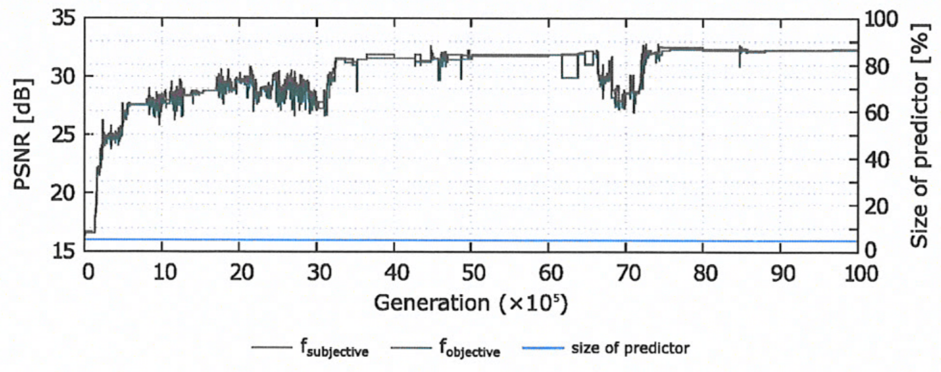
independent runs considered). A brighter pixel means that this pixel was selected with a higher frequency to the predictor. A white point denotes a situation where the fitness case represented by this pixel has been used in all fitness predictions during the course of coevolution. A black point denotes that the fitness case has never been selected.

It is interesting in Figure 12b that selected points do not focus entirely on noisy pixels (or the edges) in the training images, but they are well distributed over the image during the coevolution. If all fitness cases addressed by the predictor were focused on the interesting regions of the training image, the predictor would represent a very high error rate. However, some differences in the sample points selection have been observed between different tasks (Figs. 12a and 12c). It has to be noted that the fitness cases addressed by the fitness predictors are variable in response to the program evolution.

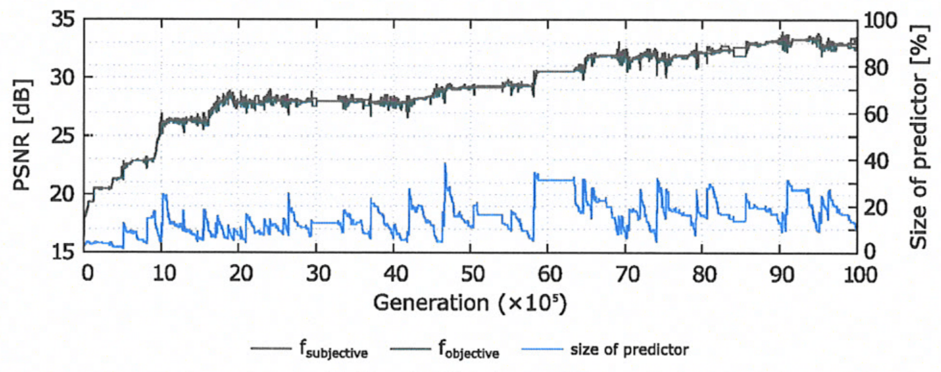
7 Conclusions

We introduced the coevolution of fitness predictors and Cartesian programs in order to accelerate CGP running on a common processor. A new method which enables us (besides useful fitness cases selection) to automatically adapt the predictor size for a given problem, and thus reduce the time required by CGP search, was proposed and evaluated.

The method was applied to five different symbolic regression tasks. Our approach outperformed the original constant-size predictors in terms of success rate and

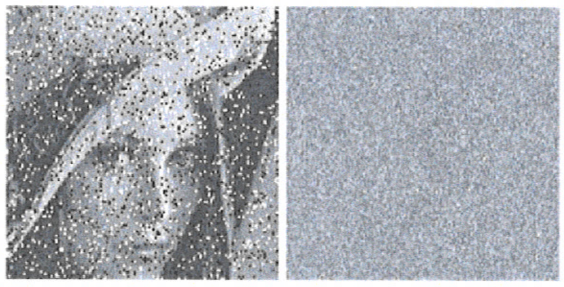


(a) The constant-size predictor containing 5% pixels.

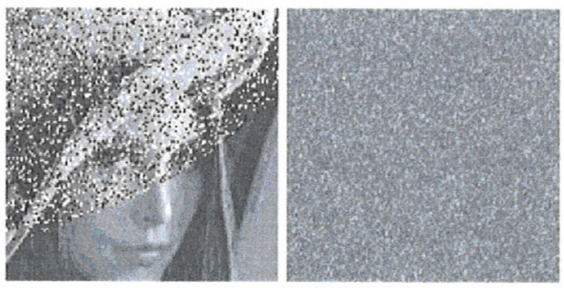


(b) The adaptive-size predictor.

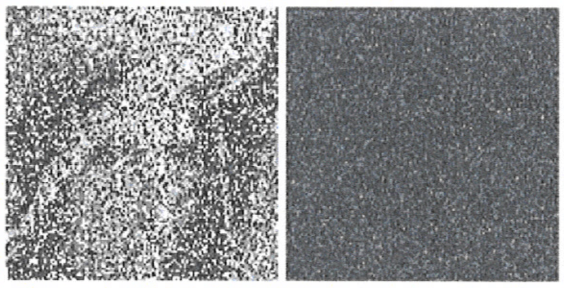
Figure 11: The progress of one run of coevolution.



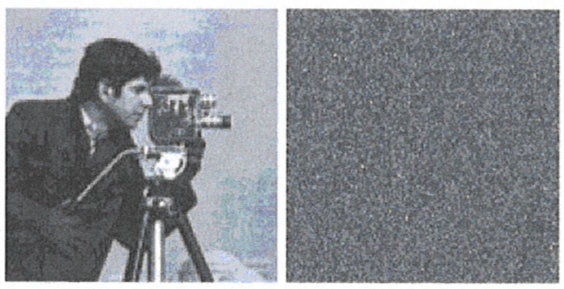
(a) Salt-and-pepper noise 15%.



(b) Partial salt-and-pepper noise 25%.



(c) Salt-and-pepper noise 50%.



(d) The edge detector.

Figure 12: The frequency of fitness cases used for the fitness prediction. (The left side shows the image and the right side shows the 2D histogram.)

computational cost. The coevolution was able to adapt the predictor size with respect to a given problem in response to the development in the candidate program evolution.

Then, the proposed method was evaluated in the task of evolutionary design of image filters of various types. It was shown that the predictor size can automatically be adapted to a given type of noise in such a way that the total evolution time is significantly reduced with respect to the standard CGP. In the case of coevolutionary CGP with the constant-size predictor, many independent runs of CGP have to be performed in order to find the most suitable predictor size for a particular task. CGP with the adaptive-size predictor eliminates the need to perform these multiple experiments. Detailed experimental analysis of the proposed method was performed. The method led to a suitable size of the predictor independent of its initial size. It was also observed that the fitness cases included in the predictor well represent the input data.

The results presented for symbolic regression problems and image filter evolution confirmed that the coevolutionary CGP, based on the adaptive-size predictor, is applicable across several domains. It enables us to significantly accelerate the search for the most suitable parameters of coevolutionary CGP.

In our future work, we plan to combine the proposed method with a hardware accelerator of filter evaluation in an embedded HW/SW system implementing adaptive video filtering.

Acknowledgments

This work was supported by The Ministry of Education, Youth and Sports of the Czech Republic from the National Programme of Sustainability (NPU II); project IT4Innovations excellence in science—LQ1602.

References

Baldwin, J. M. (1896). A new factor in evolution. *The American Naturalist*, 30(354):441–451.

De Jong, E. D., and Bucci, A. (2008). Objective set compression. In *Multiobjective Problem Solving from Nature*, pp. 357–376.

De Jong, E. D., and Pollack, J. B. (2004). Ideal evaluation from coevolution. *Evolutionary Computation*, 12(2):159–192.

Dobai, R. (2014). Evolutionary on-line synthesis of hardware accelerators for software modules in reconfigurable embedded systems. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications*, pp. 1–6.

Dolin, B., Bennett-III, F. H., and Reiffel, G. (2002). Co-evolving an effective fitness sample: Experiments in symbolic regression and distributed robot control. In *Proceedings of the 2002 ACM Symposium on Applied Computing*, pp. 553–559.

Dolinsky, J. U., Jenkinson, I. D., and Colquhoun, G. J. (2007). Aplication of genetic programming to the calibration of industrial robots. *Computers in Industry*, 58(3):255–264.

Gagné, C., and Parizeau, M. (2007). Co-evolution of nearest neighbor classifiers. *International Journal of Pattern Recognition and Artificial Inteligence*, 21(5):921–946.

Gathercole, C., and Ross, P. (1994). Dynamic training subset selection for supervised learning in genetic programming. In *Parallel Problem Solving from Nature*, pp. 312–321.

Gonzalez, R. C., Woods, R. E., and Eddins, S. L. (2009). Standard test images. *ImageProcessingPlace.com*. Retrieved from http://www.imageprocessingplace.com/

Hillis, W. D. (1990). Co-evolving parasites improve simulated evolution as an optimization procedure. *Physica D*, 42(1):228–234.

Jin, Y. (2005). A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing Journal*, 9(1):3–12.

Jin, Y., and Sendhoff, B. (2004). Reducing fitness evaluations using clustering techniques and neural network ensembles. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, pp. 688–699.

Koza, J. R. (1992). *Genetic programming: On the programming of computers by means of natural selection*, Vol. 1. Cambridge, MA: MIT Press.

Lasarczyk, C. W., Dittrich, P., and Banzhaf, W. (2004). Dynamic subset selection based on a fitness case topology. *Evolutionary Computation*, 12(2):223–242.

Mendes, R., de Voznika, F., Freitas, A., and Nievola, J. (2001). Discovering fuzzy classification rules with genetic programming and co-evolution. In *Principles of Data Mining and Knowledge Discovery*, pp. 314–325. Lecture Notes in Computer Science, Vol. 2168.

Miller, J., and Turner, A. (2015). Cartesian genetic programming. In *Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pp. 179–198.

Miller, J. F., and Thomson, P. (2000). Cartesian genetic programming. In *Genetic Programming*, pp. 121–132. Lecture Notes in Computer Science, Vol. 1802.

Monroy, G. A., Stanley, K. O., and Miikkulainen, R. (2006). Coevolution of neural networks using a layered Pareto archive. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pp. 329–336.

Nordin, P., and Banzhaf, W. (1997). An on-line method to evolve behavior and to control a miniature robot in real time with genetic programming. *Adaptive Behavior*, 5(2):107–140.

Pagie, L., and Hogeweg, P. (1997). Evolutionary consequences of coevolving targets. *Evolutionary Computation*, 5(4):401–418.

Panait, L., Luke, S., and Harrison, J. F. (2006). Archive-based cooperative coevolutionary algorithms. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pp. 345–352.

Popovici, E., Bucci, A., Wiegand, R. P., and De Jong, E. D. (2012). Coevolutionary principles. *Handbook of natural computing* (pp. 987–1033). Berlin: Springer.

Potter, M. A., and De Jong, K. A. (2000). Cooperative coevolution: An architecture for evolving coadapted subcomponents. *Evolutionary Computation*, 8(1):1–29.

Salvador, R., Otero, A., Mora, J., De La Torre, E., Riesgo, T., and Sekanina, L. (2013). Self-reconfigurable evolvable hardware system for adaptive image processing. *IEEE Transactions on Computers*, 62(8):1481–1493.

Schmidt, M. D., and Lipson, H. (2006). Co-evolving fitness predictors for accelerating and reducing evaluations. In *Genetic Programming Theory and Practice IV*, Vol. 5, pp. 113–130.

Schmidt, M. D., and Lipson, H. (2008). Coevolution of fitness predictors. *IEEE Transactions on Evolutionary Computation*, 12(6):736–749.

Schmidt, M., and Lipson, H. (2009). Distilling free-form natural laws from experimental data. *Science*, 324(5923):81–85.

Sekanina, L., Harding, S. L., Banzhaf, W., and Kowaliw, T. (2011). Image processing and CGP. *Cartesian genetic programming*, pp. 181–215. Berlin: Springer.

Shi, M. (2011). Empirical analysis of cooperative coevolution using blind decomposition. In *Proceedings of the 13th Annual Conference Companion on Genetic and Evolutionary Computation (GECCO)*, pp. 141–142.

Shi, M., and Wu, H. (2009). Pareto cooperative coevolutionary genetic algorithm using reference sharing collaboration. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO)*, pp. 867–874.

Sikulova, M., Hulva, J., and Sekanina, L. (2015). Indirectly encoded fitness predictors coevolved with Cartesian programs. In *Genetic Programming*, pp. 113–125. Lecture Notes in Computer Science, Vol. 9025.

Sikulova, M., and Sekanina, L. (2012a). Acceleration of evolutionary image filter design using coevolution in Cartesian GP. In *Parallel Problem Solving from Nature*, pp. 163–172. Lecture Notes in Computer Science, Vol. 7491.

Sikulova, M., and Sekanina, L. (2012b). Coevolution in cartesian genetic programming. In *Genetic Programming*, pp. 182–193. Lecture Notes in Computer Science, Vol. 7244.

Stanley, K. O., and Miikkulainen, R. P. (2004). *Efficient evolution of neural networks through complexification*. Retrieved from http://citeseerX.ist.psu.edu

Wiglasz, M., and Drahosova, M. (2016). Plastic fitness predictors coevolved with Cartesian programs. In *Genetic Programming*, pp. 1–16. Lecture Notes in Computer Science, Vol. 9594.

Yang, Z., Tang, K., and Yao, X. (2008). Large scale evolutionary optimization using cooperative coevolution. *Information Science*, 178(15):2985–2999.