Adaptation of a Success Story in GAs: Estimation-of-Distribution Algorithms for Tree-based Optimization Problems

Peter A.N. Bosman[1] and Edwin D. de Jong[2]

[1] Centre for Mathematics and Computer Science, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands, Peter.Bosman@cwi.nl
[2] Institute of Information and Computing Sciences, Utrecht University, P.O. Box 80.089, 3508 TB Utrecht, The Netherlands, dejong@cs.uu.nl

Summary. Fundamental research into Genetic Algorithms (GA) has led to one of the biggest successes in the design of stochastic optimization algorithms: Estimation-of-Distribution Algorithms (EDAs). These principled algorithms identify and exploit structural features of a problems structure during optimization. EDA design has so far been limited to classical solution representations such as binary strings or vectors of real values. In this chapter we adapt the EDA approach for use in optimizing problems with tree representations and thereby attempt to expand the boundaries of successful evolutionary algorithms. To do so, we propose a probability distribution for the space of trees, based on a grammar. To introduce dependencies into the distribution, grammar transformations are performed that facilitate the description of specific subfunctions. The results of performing experiments on two benchmark problems demonstrate the feasibility of the approach.

1 Introduction

Research into evolutionary computation (EC) has shown that effective optimization with evolutionary algorithms (EAs) requires an EA to have an inductive bias that can be fit to the structure of the problem under study quickly and reliably. For a fixed representation of solutions, the use of non-adaptive recombination operators such as classical crossover operators will not lead to efficient optimization unless the bias introduced by the recombination operator fits the problem's structure [28, 29]. Typically, the required resources to optimally solve the optimization problem grow exponentially with the problem size. In contrast, using the optimal crossover operator the required resources to solve the optimization problem optimally typically grow only polynomially with the problem size with the polynomial being of low order [9]. For more efficient optimization, therefore, either the representation has to be adapted to fit the bias of the recombination operator, or the operator has to be adapted to fit the structure of the optimization problem. The

P.A.N. Bosman and E.D. de Jong: *Adaptation of a Success Story in GAs: Estimation-of-Distribution Algorithms for Tree-based Optimization Problems*, Studies in Computational Intelligence (SCI) **92**, 3–18 (2008)
www.springerlink.com

latter approach is the most commonly used, that is, problem-specific recombination operators are designed. If we assume to have no prior knowledge of the problem under study, however, it is impossible to do so. The only alternative then is to expand the capacity of the model in the recombination operator so that it can be adapted to the optimization problem.

The possibility to adapt the model to the optimization problem by inducing it from the selected solutions contributes to the attractiveness of EAs. To adapt the model in a meaningful way, induction must be used on previously evaluated solutions. Because EAs work with a population of solutions, they are natural candidates for optimization based on model adaptation by induction since all kinds of statistical methods can be used that work on collections of samples, such as the estimation of probability distributions. Estimation-of-distribution algorithms (EDAs) represent one of the most successful types of EA that employ model-adaptation [13, 15, 21]. As such, EDAs can be seen as an important success in EC. EDAs provide an elegant way of bringing about an adaptive search bias. The model reflecting the bias is a probability distribution. Fitting the model to the problem's structure is done by estimating the probability distribution from the set of selected solutions. The probability distribution itself is an explicit model for the inductive search bias. Estimating the probability distribution from data corresponds to tuning the model for the inductive search bias. Because a lot is known about how probability distributions can be estimated from data [5, 14] the flexibility of the inductive search bias of EDAs is potentially large. In addition, the tuning of the inductive search bias in this fashion also has a rigorous foundation in the form of the well-established field of probability theory.

Although clearly more computational effort is required to tune the inductive search bias by estimating the probability distribution, the payoff has been shown to be worth the computational investment (e.g. polynomial scale-up behavior instead of exponential scale-up behavior and successful applications to important problems) [1, 4, 8, 17, 18, 20]. It is now interesting to investigate how this success in EC can be carried over to the domain of solution representations without a fixed length, which are typically trees. Trees are the common type of representation in genetic programming (GP), a very important area of evolutionary computation [7, 11].

GP is an important area in EC that offers algorithms to search highly expressive classes of functions, and has been applied to a diverse range of problems including circuit design, symbolic regression, and control. Basic GP employs the subtree-crossover operator, which exchanges randomly selected subtrees between individuals. Due to the particular structure of trees, subtrees (rather than e.g. any combination of nodes and arcs) appear a reasonable choice as the form of the partial solutions that will be exchanged since the functionality of a subtree is independent of its place within the tree. However, basic GP does not make informed choices as to which partial solutions will be exchanged; both the size of the removed subtree and the position where it will be inserted are chosen randomly.

Several subtree encapsulation methods exist that make more specific choices as to which partial solutions are selected to be further propagated, e.g. GLiB [2], ADFs [12], and ARL [24]. Subtree encapsulation methods have been found to substantially improve GP performance. Yet, the criteria for the selection of partial solutions they employ are still heuristic; typically, either the fitness of the tree in which a subtree occurs is used as an indication of its value, or the partial solution is itself subject to evolution.

In this chapter we explore how the success of EDAs may be used in GP. We investigate whether a more principled approach to the selection of partial solutions and their recombination is possible. If the distribution of high-fitness trees can be estimated, this would directly specify which combinations of elements are to be maintained in the creation of new individuals and thus which combinations of partial solutions may be fruitfully explored. We investigate how the principle of distribution estimation may be employed in the context of tree-based problems. In GAs, the development of EDAs and other linkage learning techniques has yielded a better insight into the design of competent GAs by rendering the assumptions implicitly made by algorithms explicit. Our aim is that the application of distribution estimation techniques to tree-based problems may likewise clarify the design of principled methods for GP. This chapter represents a first step in that direction.

To estimate distributions over the space of trees, a representation must be chosen. In Probabilistic Incremental Program Evolution (PIPE) [25], trees are matched on a fixed-size template, such that the nodes becomes uniquely identifiable variables. The size of the template may increase over time however as trees grow larger. Using this representation, all nodes are treated equally. While this permits the encapsulation of any combination of nodes, it does not exploit the particular non-fixed-size and variable–child–arity structure of trees. The complexity of such an algorithm is determined by the maximally allowed shape for the trees, which must be chosen in advance. In PIPE the distribution is the same for each node, which corresponds to a univariately factorized probability distribution over all nodes. In Extended Compact Genetic Programming (ECGP) [26], trees are represented in the same way as in PIPE. However, the probability distribution that is estimated is a marginal-product factorization that allows the modelling of dependencies between multiple nodes that are located anywhere in the fixed-size template. Moreover, the size of the template is fixed beforehand in ECGP and does not increase over time. In this chapter, a method will be employed that estimates the distribution of trees based on the subtrees that actually occur. The representation specifies a set of rules whose expansion leads to trees. The rules capture local information, thereby offering a potential to exploit the specific structure of trees, while at the same time their use in an EDA offers a potential for generalization that is not provided by using fixed-size templates. Shan et al. [27] use a similar approach (based on stochastic grammars). The main difference is that their approach starts from an explicit description of a specific set of trees and then generalizes the description to represent common subtrees.

In our approach we do the reverse. Our approach starts from a description of all possible trees and then tunes it to more specifically describe the set of trees.

The remainder of this chapter is organized as follows. In Sect. 2 we introduce basic terminology. In Sect. 3 we define a probability distribution over trees. We also show how to draw new samples from the distribution and how the distribution is estimated from data. In Sect. 4 we perform experiments on two benchmark problems and compare the performance of standard GP and an EDA based on the proposed distribution. We conclude this chapter in Sect. 5.

2 Terminology

A grammar $\boldsymbol{G}$ is a vector of l_r production rules $R_j, j \in \{0, 1, \ldots, l_r - 1\}$, that is, $\boldsymbol{G} = (R_0, R_1, \ldots, R_{l_r-1})$.

A production rule is denoted by $R_j : \mathcal{S}_k \rightarrow E_j$, where $\mathcal{S}_k$ is a symbol that can be replaced with the expansion E_j of the production rule. Let K be the number of available symbols, then $k \in \{0, 1, \ldots, K-1\}$. We will use only one symbol and allow ourselves to write $\mathcal{S}$ instead of $\mathcal{S}_k$.

An expansion E_j of production rule R_j is a tree. We will therefore generally call E_j an expansion tree. The internal nodes of an expansion tree are functions with at least one argument. The leaves are either symbols, constants or input variables. An example of an expansion tree is $E_j = +(\sin(\mathcal{S})), (-(\log(\mathcal{S}), \cos(\mathcal{S})))$ which, in common mathematical notation, represents $\sin(\mathcal{S}) + (\log(\mathcal{S}) - \cos(\mathcal{S}))$.

A sentence is obtained from a grammar if a production rule is chosen to replace a symbol repeatedly until all symbols have disappeared. Sentences can therefore be seen as trees. We denote a sentence by s. We will denote a subtree of a sentence by t and call it a sentence subtree.

A sentence subtree t is said to be matched by an expansion tree E_j, denoted $t \in E_j$ if and only if all nodes of E_j coincide with nodes found in t following the same trails, with the exception of symbol nodes, which may be matched to any non-empty sentence subtree.

The following grammar $\boldsymbol{G} = (R_0, R_1, \ldots, R_{5+n})$ with $l_r = 6 + n$ is an example of a grammar that describes certain n-dimensional real-valued functions:

$$
\begin{array}{ll}
R_0 : \mathcal{S} \rightarrow c \text{ (a constant} \in \mathbb{R}) & R_{n+1} : \mathcal{S} \rightarrow +(\mathcal{S}, \mathcal{S}) \\
R_1 : \mathcal{S} \rightarrow i_0 \text{ (input variable 0)} & R_{n+2} : \mathcal{S} \rightarrow \cdot(\mathcal{S}, \mathcal{S}) \\
R_2 : \mathcal{S} \rightarrow i_1 \text{ (input variable 1)} & R_{n+3} : \mathcal{S} \rightarrow -(\mathcal{S}) \\
\vdots \quad \vdots & R_{n+4} : \mathcal{S} \rightarrow \sin(\mathcal{S}) \\
R_n : \mathcal{S} \rightarrow i_{n-1} \text{ (input variable n–1)} & R_{n+5} : \mathcal{S} \rightarrow \cos(\mathcal{S})
\end{array}
$$

3 Probability Distribution

3.1 Definition

To construct a probability distribution over sentences, we introduce a random variable S that represents a sentence and a random variable T that represents a sentence subtree. Because sentence subtrees are recursive structures we define a probability distribution for sentence subtrees recursively. To do so, we must know where the tree terminates. This information can be obtained by taking the depth of a sentence subtree into account. Let $P^{\boldsymbol{G}}(T = t|D = d)$ be a probability distribution over all sentence subtrees t that occur at depth d in a sentence. Now, we define the probability distribution over sentences s by:

$$P^{\boldsymbol{G}}(S = s) = P^{\boldsymbol{G}}(T = s|D = 0) \tag{1}$$

Since sentence subtrees are constructed using production rules, we can define $P^{\boldsymbol{G}}(T = t|D = d)$ using the production rules. Since there is only one symbol, we can also focus on the expansion trees. Although depth can be used to model the probability of terminating a sentence at some node in the tree, sentences can be described more precisely if depth is also used to model the probability of occurrence of functions at specific depths. Preliminary experiments indicated that this use of depth information leads to better results.

We define $P_j^E(J = j|D = d)$ to be a discrete conditional probability distribution that models the probability of choosing expansion tree E_j, $j \in \{0, 1, \ldots, l_r\}$ at depth d when constructing a new sentence.

We assume that the values of the constants and the indices of the input variables in an expansion tree are not dependent on the depth. Conforming to this assumption we define $2l_r$ multivariate probability distributions that allow us to model the use of constants and inputs inside production rules other than the standard rules $\mathcal{S} \to c$ and $\mathcal{S} \to i_k$, $k \in \{0, 1, \ldots, n-1\}$:

- $P_j^C(\boldsymbol{C}_j)$, $j \in \{0, 1, \ldots, l_r - 1\}$, a probability distribution over all constants in expansion tree E_j, where $\boldsymbol{C}_j = (C_{j0}, C_{j1}, \ldots, C_{j(n_{C_j}-1)})$. Each C_{jk} is a random variable that represents a constant in E_j.
- $P_j^I(\boldsymbol{I}_j)$, $j \in \{0, 1, \ldots, l_r - 1\}$, a probability distribution over all inputs in expansion tree E_j, where $\boldsymbol{I}_j = (I_{j0}, I_{j1}, \ldots, I_{j(n_{I_j}-1)})$. Each I_{jk} is a random variable that represents input variables, i.e. $I_{jk} \in \{0, 1, \ldots, n-1\}$.

The definition of $P_j^I(\boldsymbol{I}_j)$ enforces a single production rule $\mathcal{S} \to i$, where i represents all input variables, instead of n production rules $\mathcal{S} \to i_j$, $j \in \{0, 1, \ldots, n-1\}$. This reduces the computational complexity for estimating the probability distribution, especially if n is large. However, it prohibits the introduction of production rules that make use of specific input variables.

We will enforce that any sentence subtree t can be matched by only one expansion tree E_j. The probability distribution over all sentence subtrees at

some given depth $D = d$ is then the product of the probability of matching the sentence subtree with some expansion tree E_j and the product of all (recursive) probabilities of the sentence subtrees located at the symbol–leaf nodes in E_j.

Let $\mathcal{S}_{jk}$ be the kth symbol in expansion tree E_j, $k \in \{0, 1, \ldots, n_{\mathcal{S}_j} - 1\}$. Let $stree(\mathcal{S}_{jk}, t)$ be the sentence subtree of sentence subtree t at the same location where $\mathcal{S}_{jk}$ is found in the expansion tree E_j that matches t. Let $depth(\mathcal{S}_{jk})$ be the depth of $\mathcal{S}_{jk}$ in expansion tree E_j. Let $match(t)$ be the index of the matched expansion tree, i.e. $match(t) = j \Leftrightarrow t \in E_j$. We then have:

$$P^{\boldsymbol{G}}(T = t|D = d) = \tag{2}$$

$$P^E(J = j|D = d)P_j^C(\boldsymbol{C}_j)P_j^I(\boldsymbol{I}_j)\prod_{k=0}^{n_{\mathcal{S}_j}-1} P^{\boldsymbol{G}}(T = stree(\mathcal{S}_{jk}, t)|D = d+depth(\mathcal{S}_{jk}))$$

$$\text{where } j = match(t)$$

Summarizing, formulating a probability distribution for sentences requires:

- $P^E(J|D)$; a univariate discrete conditional probability distribution over all possible production rules, conditioned on the depth of occurrence.
- $P_j^C(\boldsymbol{C}_j)$; l_r multivariate probability distributions over n_{C_j} variables.
- $P_j^I(\boldsymbol{I}_j)$; l_r multivariate probability distributions over n_{I_j} variables.

3.2 Estimation from Data

General Greedy Approach

We choose a greedy approach to estimate $P^{\boldsymbol{G}}(S)$ from a set $\boldsymbol{\mathcal{S}}$ of sentences. Greedy approaches have been used in most EDAs so far and have led to good results [3, 13, 16, 19]. The algorithm starts from a given initial grammar. Using transformations, the grammar is made more involved. A transformation results in a set of candidate grammars, each slightly different and each more involved than the current grammar. The goodness of each candidate grammar is then evaluated and the best one is accepted if it is better than the current one. If there is no better grammar, the greedy algorithm terminates.

Transformations

The only transformation that we allow is the substitution of one symbol of an expansion tree with one expansion tree from the base grammar. The base grammar is the grammar that is initially provided. To ensure that after a transformation any sentence subtree can be matched by only one expansion

tree we assume that the base grammar has this property. To transform the grammar we only allow to substitute symbols with expansion trees of the base grammar. Here is an example of expanding the base grammar (first column) using a single expansion (second column):

Base grammar	**Single expansion**	**Full expansion**
$R_0 : \mathcal{S} \to c$	$R_0 : \mathcal{S} \to c$	$R_0 : \mathcal{S} \to c$
$R_1 : \mathcal{S} \to i$	$R_1 : \mathcal{S} \to i$	$R_1 : \mathcal{S} \to i$
$R_2 : \mathcal{S} \to f(\mathcal{S}, \mathcal{S})$	$R_2 : \mathcal{S} \to f(\mathcal{S}, \mathcal{S})$	$R_2 : \mathcal{S} \to f(\mathcal{S}, \mathcal{S})$
$R_3 : \mathcal{S} \to g(\mathcal{S}, \mathcal{S})$	$R_3 : \mathcal{S} \to g(\mathcal{S}, \mathcal{S})$	$R_3 : \mathcal{S} \to g(c, \mathcal{S})$
	$R_4 : \mathcal{S} \to g(f(\mathcal{S}, \mathcal{S}), \mathcal{S})$	$R_4 : \mathcal{S} \to g(i, \mathcal{S})$
		$R_5 : \mathcal{S} \to g(f(\mathcal{S}, \mathcal{S}), \mathcal{S})$
		$R_6 : \mathcal{S} \to g(g(\mathcal{S}, \mathcal{S}), \mathcal{S})$

Note that the set of expansion trees can now no longer be matched uniquely to all sentence trees. For instance, sentence $g(f(c, c), c)$ can at the top level now be matched by expansion trees 3 and 4. To ensure only a single match, we could expand every expansion tree from the base grammar into a production rule and subsequently remove the original production rule that has now been expanded (third column in the example above). However, this rapidly increases the number of production rules in the grammar and may introduce additional rules that aren't specifically interesting for modelling the data at hand. To be able to only introduce the rules that are interesting, we equip the symbols with a list of indices that indicate which of the production rules in the base grammar may be matched to that symbol. Once a substitution occurs, the symbol that was instantiated may no longer match with the expansion tree that was inserted into it. For example:

Base grammar	**Expanded grammar**
$R_0 : \mathcal{S} \to c$	$R_0 : \mathcal{S} \to c$
$R_1 : \mathcal{S} \to i$	$R_1 : \mathcal{S} \to i$
$R_2 : \mathcal{S} \to f(\mathcal{S}^{0,1,2,3}, \mathcal{S}^{0,1,2,3})$	$R_2 : \mathcal{S} \to f(\mathcal{S}^{0,1,2,3}, \mathcal{S}^{0,1,2,3})$
$R_3 : \mathcal{S} \to g(\mathcal{S}^{0,1,2,3}, \mathcal{S}^{0,1,2,3})$	$R_3 : \mathcal{S} \to g(\mathcal{S}^{0,1,3}, \mathcal{S}^{0,1,2,3})$
	$R_4 : \mathcal{S} \to g(f(\mathcal{S}^{0,1,2,3}, \mathcal{S}^{0,1,2,3}), \mathcal{S}^{0,1,2,3})$

A sentence can now be preprocessed bottom-up in $\mathcal{O}(n)$ time to indicate for each node which expansion tree matches that node, where n is the number of nodes in the tree. The sentence can then be traversed top-down to perform the frequency count for the expansion trees. It should be noted that this approach means that if a symbol list does not contain all indices of the base grammar, then it represents only the set of the indicated rules from the base grammar. In the above example for instance $\mathcal{S}^{0,1,3}$ represents $\mathcal{S} \to c$, $\mathcal{S} \to i$ and $\mathcal{S} \to g(\mathcal{S}^{0,1,2,3}, \mathcal{S}^{0,1,2,3})$. Therefore, the probability associated with this particular symbol is not the recursive application of the distribution in (2), but is uniform over the indicated alternatives. This is comparable to the approach

of default tables for discrete random variables in which instead of indicating a probability for all possible combinations of values for the random variables, only a subset of them is explicitly indicated. All remaining combinations are assigned an equal probability such that the distribution sums to one over all possible values.

Goodness Measure

The goodness of a grammar is determined using its associated probability distribution. This distribution can be estimated by traversing each sentence and by computing the proportions of occurrence for the expansion trees at each depth. Probability distributions $P_j^C(\boldsymbol{C}_j)$ and $P_j^I(\boldsymbol{I}_j)$ can be estimated after filtering the sentences to obtain for each expansion tree a set of sets of constants and a set of sets of variables (one set of constants or variables for each match of the expansion tree). From these sets, the distributions $P_j^C(\boldsymbol{C}_j)$ and $P_j^I(\boldsymbol{I}_j)$ can be estimated using well-known density estimation techniques (such as proportion estimates for the discrete input variable indices).

Once all parameters have been estimated, the probability-distribution value of each sentence in the data can be computed to obtain the likelihood (or the negative log-likelihood). The goodness measure that we ultimately use to distinguish between probability distributions is the MDL metric [23], which is a form of likelihood penalization, also known as the extended likelihood principle [6]:

$$\text{MDL}\left(P^{\boldsymbol{G}}(S)\right) = -\sum_{i=0}^{|\boldsymbol{\mathcal{S}}|-1} \ln\left(P^{\boldsymbol{G}}(S=\boldsymbol{\mathcal{S}}_i)\right) + \frac{1}{2}\ln(|\boldsymbol{\mathcal{S}}|)|\boldsymbol{\theta}| + \zeta \qquad (3)$$

where $|\boldsymbol{\theta}|$ denotes the number of parameters that need to be estimated in probability distribution $P^{\boldsymbol{G}}(T)$ and ζ denotes the expected additional number of bits required to store the probability distribution. In our case, $|\boldsymbol{\theta}|$ equals:

$$|\boldsymbol{\theta}| = 2d_{\max} + l_r - 4 + \left(\sum_{j=0}^{l_r-1} |\boldsymbol{\theta}| \leftarrow P_j^C(\boldsymbol{C}_j)\right) + \left(\sum_{j=0}^{l_r-1} n^{n_{I_j}} - 1\right) \qquad (4)$$

Since in the latter case, the number of parameters grows exponentially with the number of input variables in an expansion tree, this is likely to be a strong restriction on the number of input variables that the greedy algorithm will allow into any expansion tree. As an alternative, independence can be enforced between input variables. The resulting distribution can then no longer model correlation between multiple input variables in one expansion tree. The number of parameters to be estimated reduces drastically however and this in turn will allow more input variables to be incorporated into the

expansion trees. Effectively, we thus enforce $P_j^I(\boldsymbol{I}_j) = \prod_{k=0}^{n_{I_j}-1} P_j^I(I_{jk})$, which changes the last term in the number of parameters $|\boldsymbol{\theta}|$ into $\sum_{j=0}^{l_r-1}(n-1)n_{I_j}$.

The expected additionally required number of bits ζ in our case come from the production rules. The more production rules, the more bits are required. Moreover, longer production rules require more bits. The number of bits required to store a production rule is $\ln(l_{r_{base}})$ times the number of internal nodes in the expansion tree where $l_{r_{base}}$ is the number of production rules in the base grammar. Finally, symbols have lists of production-rule indices. To store one such list, the number of required bits equals $\ln(l_{r_{base}})$ times the length of that list. The value of ζ equals the sum of these two terms over all production rules.

3.3 Sampling

1. Initialize the sentence tree of the new sentence: introduce a single symbol as the root.
2. Set $d = 0$.
3. Visit the symbol at the root.
4. Draw an integer $j \in \{0, 1, \ldots, l_r - 1\}$ randomly from $P^E(J|D = d)$ if the list of production-rule indices associated with the visited symbol is complete, or draw a random integer from the list otherwise.
5. Make a copy of expansion tree E_j.
6. Draw all constants in the copied tree randomly from $P_j^C(\boldsymbol{C}_j)$.
7. Draw all input indices in the copied tree randomly from $P_j^I(\boldsymbol{I}_j)$.
8. Expand the current sentence tree by replacing the symbol that is being visited with the copied tree.
9. Extract the symbols from the copied tree and their relative depths.
10. Recursively visit each symbol in the copied tree after setting the depth d properly (current depth plus relative symbol depth).

4 Experiments

We have performed experiments with our approach to estimating a probability distribution over sentences by using it in an EDA. We applied the resulting EDA as well as a standard GP algorithm that only uses subtree-swapping crossover to two GP–benchmarking problems, namely the Royal Tree Problem by Punch, Zongker and Goodman [22] and the tunable benchmark problem by Korkmaz and Üçoluk [10], which we will refer to as the binary-functions problem. Both problems are non-functional, which means that fitness is defined completely in terms of the structure of the sentence and input variables have an empty domain.

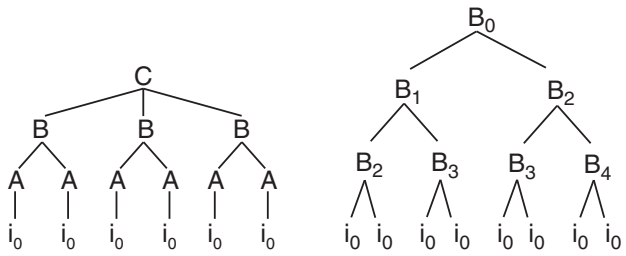

Fig. 1. Examples of sentences for the Royal Tree Problem (*left*) and the binary-functions problem (*right*)

4.1 Benchmark Problems

In the Royal Tree Problem, there is one function of arity n for each $n \in \{0, 1, 2, \ldots\}$. The function with arity 0 is i_0. The other functions are labeled A for arity 1, B for arity 2 and so on. The fitness function is defined recursively and in terms of "perfect" subtrees. A subtree is perfect if it is a full tree, i.e. all paths from the root to any leaf are equally long and the arity of any function is $n-1$ where n is the arity its parent function. Figure 1 gives an example of a perfect tree of height 4. The fitness of a sentence is the fitness of the root. The fitness of a node is a weighted sum of the fitness values of its children. The weight of a child is

- *Full*
  if the child is a perfect subtree with a root function of arity $n-1$ where n is the arity of the function in the current node.
- *Partial*
  if the child is not a perfect subtree but it has the correct root function.
- *Penalty*
  otherwise.

If the node is itself the root of a perfect tree, the final sum is multiplied by *Complete*. In our experiments, we follow the choices of Punch, Zongker and Goodman [22] and set *Full* $= 2$, *Partial* $= 1$, *Penalty* $= \frac{1}{3}$ and *Complete* $= 2$. The structure of this problem lies in perfect subtrees and child nodes having a function of arity that is one unit smaller than the function of their parent. Coincidentally, this results in a very strong dependence on the depth, since the optimal sentence subtree of height d has one function of arity $d-1$ in the root, $d-1$ functions of arity $d-2$ on the second level and so on (see Fig. 1).

In the binary-functions problem there are various functions B_i, all of which have arity two. Similar to the Royal Tree Problem, there is only one input variable i_0. The fitness of a sentence is again the fitness of the root. The fitness of a node is 1 if the node is an input variable. Otherwise the fitness is a

combination of the fitness values of its two children C_1 and C_2; it is computed according to:

$$\begin{cases} f(C_1) + f(C_2) & \text{if both children are inputs or no constraints violated} \\ \eta f(C_1) + f(C_2) & \text{if } C_1 \text{ violates first constraint, } C_2 \text{ doesn't violate first constraint and second constraint not violated} \\ f(C_1) + \eta f(C_2) & \text{if } C_2 \text{ violates first constraint, } C_1 \text{ doesn't violate first constraint and second constraint not violated} \\ \eta f(C_1) + \eta f(C_2) & \text{if both children violate first constraint or the second constraint is violated} \end{cases}$$

where the indicated constraints are:

1. The index of the parent function is smaller than the indices of its children.
2. The index of the left child is smaller than the index of the right child.

In our experiments we have used $\eta = 0.25$. The structure of this optimization problem is both local and overlapping. The local aspect lies in the fact that only a node and its direct descendants are dependent on each other. However, since these dependencies hold for each node, the structure is recursive throughout the tree and thus the dependencies are inherently overlapping.

4.2 Results

For both problems, we have computed convergence graphs for both algorithms and three different population sizes. The curves in the graphs are averaged over 25 runs. For the Royal Tree problem (see Fig. 2, left), GP achieves reasonable results for population sizes 500 and 1,000, but does not reliably find the optimum. The EDA identifies the optimal solution in every single run and for all population sizes. The Royal Tree problem is a benchmark problem for GP that features a depth-wise layered structure. Clearly, the use of depth

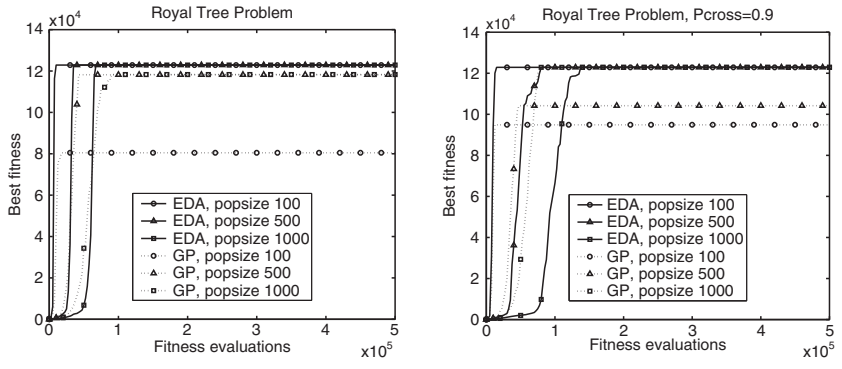

Fig. 2. Results for the Royal Tree problem

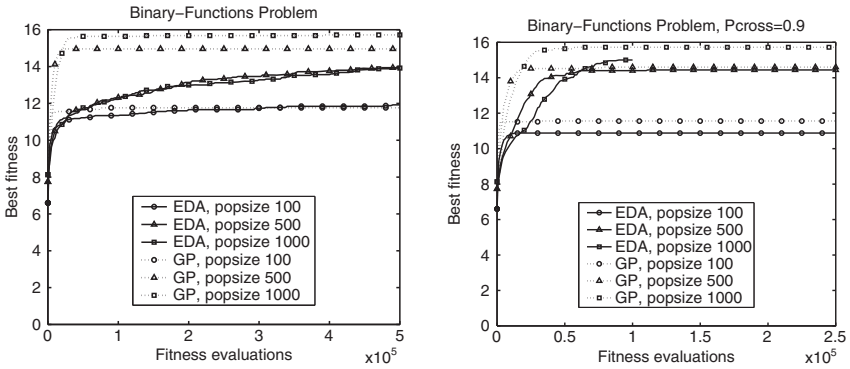

Fig. 3. Results for the binary-functions problem

information therefore renders the EDA particularly appropriate for problems of this kind. Still, this result demonstrates the feasibility of our EDA to GP. Figure 2 (right) shows the results when recombination is performed in only 90% of the cases, and copying a random parent is performed in the remaining 10%. Although GP is in this case also able to reliably find the optimum this is only the case for a population size of 1,000 whereas the EDA is still able to reliably find the optimum for all population sizes. Although lowering the probability of recombination normally speeds up convergence, in this case the EDA is only hampered by it because the distribution can perfectly describe the optimum and therefore using the distribution more frequently will improve performance. Moreover, copying introduces spurious dependencies that are estimated in the distribution and will additionally hamper optimization.

Figure 3 shows the results for the binary-functions problem (maximum fitness is 16). This problem has dependencies that are much harder for the distribution to adequately represent and reliably reproduce. Very large subfunctions are required to this end. Moreover, the multiple (sub)optima slow down convergence for the EDA as can be seen in Fig. 3 on the left. Crossover in GP is much more likely to reproduce large parts of parents. Hence, crossover automatically biases the search towards one of these solutions, allowing for faster convergence. The use of copying leads to faster convergence of the EDA. However, the deficiency of the distribution with respect to the dependencies in the problem still hamper the performance of the EDA enough to not be able to improve over standard GP. Additional enhancements may be required to make the distribution used more suited to cope with dependencies such as those encountered in the binary-functions problem, after which an improvement over standard GP may be expected similar to the improvement seen for the Royal Tree problem.

4.3 Examples of Learned Production Rules

In Fig. 4 the learned probability distributions during a run of the EDA on the Royal Tree problem are presented in a tabulated fashion. The figure shows the distributions at the beginning, in generation 50, and at the end of the run. A population of size 100 was used and the probability of recombination was set to 1.0. Clearly, the distribution can already be made specific enough by setting the probabilities because of the perfect dependency on the depth of the Royal Tree problem. Hence, no new production rules are introduced into the grammar at any point during the run and the probabilities converge towards a configuration that uniquely describes the optimal sentence.

	D					
	0	1	2	3	4	5
i_0	0.167	0.167	0.167	0.167	0.167	1.000
$A(\mathcal{S})$	0.167	0.167	0.167	0.167	0.167	0.000
$B(\mathcal{S},\mathcal{S})$	0.167	0.167	0.167	0.167	0.167	0.000
$C(\mathcal{S},\mathcal{S},\mathcal{S})$	0.167	0.167	0.167	0.167	0.167	0.000
$D(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	0.167	0.167	0.167	0.167	0.167	0.000
$E(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	0.167	0.167	0.167	0.167	0.167	0.000

	D					
	0	1	2	3	4	5
i_0	0.000	0.000	0.029	0.140	0.158	1.000
$A(\mathcal{S})$	0.000	0.000	0.027	0.131	0.318	0.000
$B(\mathcal{S},\mathcal{S})$	0.000	0.000	0.079	0.362	0.146	0.000
$C(\mathcal{S},\mathcal{S},\mathcal{S})$	0.000	0.000	0.642	0.143	0.121	0.000
$D(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	0.000	0.992	0.011	0.106	0.133	0.000
$E(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	1.000	0.008	0.011	0.116	0.123	0.000

	D					
	0	1	2	3	4	5
i_0	0.000	0.000	0.000	0.000	0.000	1.000
$A(\mathcal{S})$	0.000	0.000	0.000	0.000	1.000	0.000
$B(\mathcal{S},\mathcal{S})$	0.000	0.000	0.000	1.000	0.000	0.000
$C(\mathcal{S},\mathcal{S},\mathcal{S})$	0.000	0.000	1.000	0.000	0.000	0.000
$D(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	0.000	1.000	0.000	0.000	0.000	0.000
$E(\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S},\mathcal{S})$	1.000	0.000	0.000	0.000	0.000	0.000

	D				
	0	1	2	3	4
i_0	0.000	0.000	0.185	0.346	1.000
$B_0(\mathcal{S},\mathcal{S})$	0.082	0.000	0.020	0.000	0.000
$B_1(\mathcal{S},\mathcal{S})$	0.002	0.000	0.009	0.000	0.000
$B_2(\mathcal{S},\mathcal{S})$	0.002	0.002	0.027	0.000	0.000
$B_3(\mathcal{S},\mathcal{S})$	0.006	0.011	0.059	0.000	0.000
$B_4(\mathcal{S},\mathcal{S})$	0.001	0.097	0.106	0.110	0.000
$B_5(\mathcal{S},\mathcal{S})$	0.000	0.158	0.105	0.185	0.000
$B_6(\mathcal{S},\mathcal{S})$	0.000	0.000	0.000	0.000	0.000
$B_0(B_1(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.582	0.000	0.032	0.000	0.000
$B_0(i_0,\mathcal{S})$	0.000	0.000	0.004	0.000	0.000
$B_6(i_0,\mathcal{S})$	0.000	0.412	0.114	0.000	0.000
$B_6(\mathcal{S},i_0)$	0.000	0.295	0.121	0.000	0.000
$B_6(i_0,i_0)$	0.000	0.000	0.000	0.249	0.000
$B_1(B_2(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.189	0.000	0.034	0.000	0.000
$B_1(i_0,\mathcal{S})$	0.000	0.000	0.001	0.018	0.000
$B_3(i_0,\mathcal{S})$	0.000	0.013	0.092	0.000	0.000
$B_3(i_0,i_0)$	0.000	0.000	0.000	0.058	0.000
$B_2(i_0,\mathcal{S})$	0.000	0.003	0.026	0.000	0.000
$B_2(B_3(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.018	0.010	0.046	0.000	0.000
$B_2(i_0,i_0)$	0.000	0.000	0.000	0.019	0.000
$B_0(B_2(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.096	0.000	0.011	0.000	0.000
$B_1(B_3(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.023	0.000	0.006	0.000	0.000
$B_0(B_6(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.000	0.000	0.000	0.000	0.000
$B_0(B_0(\mathcal{S},\mathcal{S}),\mathcal{S})$	0.000	0.000	0.000	0.000	0.000
$B_0(i_0,i_0)$	0.000	0.000	0.000	0.015	0.000

Fig. 4. The probability distribution at the beginning (*top left*), after 50 generations (*center left*) and upon convergence after 139 generations (*bottom left*) of an example run of the EDA on the Royal Tree problem. Also, the probability distribution is shown after 35 generations in an example run of the EDA on the binary-functions problem (*right*). For brevity, only the expansion trees E_i of each production rule are shown and the production rule index lists are dropped for the symbols $\mathcal{S}$

In Fig. 4 the learned probability distribution in generation 35 during a run of the EDA on the binary-functions problem is also presented in a tabulated fashion. The population size used in this experiment is 10,000 and the probability of recombination is set to 0.9. In generation 35, the average fitness was 10.1379 whereas the best fitness was 13 (optimal fitness is 16). Clearly in this case the probability distribution based on the initial grammar cannot be made specific enough to describe the features of good sentences nor can it perfectly describe the optimal sentence. Therefore, additional production rules are added by the greedy search algorithm. As can be seen, the most important rules are added first and they have a high probability for at least one depth. The rules that are added afterwards mainly serve to make the distribution more specific by reducing the size of the default tables that have been created by the substitution transformations. The rules themselves have only a very low probability. One of the most important rules regarding the root of the tree has already been established (i.e. $B_0(B_1(\mathcal{S},\mathcal{S}),\mathcal{S})$). Although other important rules have been discovered, they have not yet received high probabilities as they do not occur frequently in the population. As the search progresses towards the optimum, other important rules such as $B_2(B_3(\mathcal{S},\mathcal{S}),\mathcal{S})$ will obtain more prominent probabilities as well to more specifically describe a smaller subset of high-quality sentences.

5 Discussion and Conclusions

In this chapter we have proposed a probability distribution over trees to be used in an EDA for GP. The distribution basically associates a probability with each production rule in a context-free grammar. More involved production rules or subfunctions can be introduced using transformations in which one production rule is expanded into another production rule. This allows the probability distribution to become more specific and to express a higher order of dependency. We have performed experiments on two benchmark problems from the literature. The results indicate that our EDA for GP is feasible. It should be noted however that learning advanced production rules using the greedy algorithm proposed in this chapter can take up a lot of time, especially if the number of production rules and the arity of the subfunctions increase. To speed up this greedy process, only a single rule can be randomly selected into which to expand each production rule from the base grammar instead of expanding each production rule from the base grammar into each currently available production rule. Although this significantly reduces the number of candidate distributions in the greedy algorithm, it also significantly improves the running time. Moreover, since the greedy algorithm is iterative and the probability distribution is estimated anew each generation, the most important subfunctions are still expected to emerge.

Because our approach to estimating probability distributions over trees does not fix or bound the tree structure beforehand, our approach can be

seen as a more principled way of identifying arbitrarily-sized important subfunctions than by constructing subfunctions randomly (e.g. GLiB [2]) or by evolving them (e.g. ADFs [12] and ARL [24]). The use of the EDA principle may therefore offer an exiting new direction for GP that allows to detect and exploit substructures in a more principled manner for enhanced optimization performance, thus paving the way for novel future successes in EC.

References

1. T. Alderliesten, P. A. N. Bosman, and W. Niessen. Towards a real-time minimally–invasive vascular intervention simulation system. *IEEE Transactions on Medical Imaging*, 26(1):128–132, 2007.
2. P. J. Angeline and J. B. Pollack. The evolutionary induction of subroutines. In *Proceedings of the Fourteenth Annual Conference of the Cognitive Science Society*, pages 236–241, Lawrence Erlbaum, Hillsdale, NJ, 1992.
3. P. A. N. Bosman. *Design and Application of Iterated Density–Estimation Evolutionary Algorithms*. PhD thesis, Utrecht University, Utrecht, The Netherlands, 2003.
4. P. A. N. Bosman, J. Grahl, and F. Rothlauf. SDR: A better trigger for adaptive variance scaling in normal edas. In D. Thierens et al., (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO–2007*, pages 492–499, ACM, New York, 2007.
5. W. Buntine. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*, 2:159–225, 1994.
6. W. Buntine. A guide to the literature on learning probabilistic networks from data. *IEEE Transactions on Knowledge and Data Engineering*, 8:195–210, 1996.
7. N. L. Cramer. A representation for the adaptive generation of simple sequential programs. In John J. Grefenstette, (ed.), *Proceedings of the First International Conference on Genetic Algorithms and their Applications*, Carnegie-Mellon University, pages 183–187, Lawrence Erlbaum, Hillsdale, NJ, 1985.
8. R. Etxeberria and P. Larrañaga. Global optimization using Bayesian networks. In A. A. O. Rodriguez et al., (eds.), *Proceedings of the Second Symposium on Artificial Intelligence CIMAF–1999*, pages 332–339. Institute of Cybernetics, Mathematics and Physics, 1999.
9. G. Harik, E. Cantú-Paz, D. E. Goldberg, and B. L. Miller. The gambler's ruin problem, genetic algorithms, and the sizing of populations. *Evolutionary Computation*, 7(3):231–253, 1999.
10. E. E. Korkmaz and G. Üçoluk. Design and usage of a new benchmark problem for genetic programming. In *Proceedings of the 18th International Symposium on Computer and Information Sciences ISCIS–2003*, pages 561–567, Springer, Berlin Heidelberg New York, 2003.
11. J. R. Koza. *Genetic Programming*. MIT, Cambridge, MA, 1992.
12. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT, Cambridge, MA, 1994.
13. P. Larrañaga and J. A. Lozano. *Estimation of Distribution Algorithms. A New Tool for Evolutionary Computation.* Kluwer, London, 2001.
14. S. L. Lauritzen. *Graphical Models*. Clarendon, Oxford, 1996.

15. J. A. Lozano, P. Larrañaga, I. Inza, and E. Bengoetxea. *Towards a New Evolutionary Computation*. Springer, Berlin Heidelberg New York, 2006.
16. M. Pelikan. *Bayesian optimization algorithm: From single level to hierarchy*. PhD thesis, University of Illinois at Urbana–Champaign, Urbana, Illinois, 2002.
17. M. Pelikan and D. E. Goldberg. Hierarchical BOA solves ising spin glasses and maxsat. In E. Cantú-Paz et al., (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO–2003*, pages 1271–1282, Springer, Berlin Heidelberg New York, 2003.
18. M. Pelikan, D. E. Goldberg, and E. Cantú-Paz. BOA: The Bayesian optimization algorithm. In W. Banzhaf et al., (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO–1999*, pages 525–532, Morgan Kauffman, San Francisco, California, 1999.
19. M. Pelikan, D. E. Goldberg, and F. G. Lobo. A survey of optimization by building and using probabilistic models. *Computational Optimization and Applications*, 21(1):5–20, 2002.
20. M. Pelikan, D. E. Goldberg, and K. Sastry. Bayesian optimization algorithm, decision graphs and occam's razor. In L. Spector et al., (eds.), *Proceedings of the Genetic and Evolutionary Computation Conference – GECCO–2001*, pages 519–526, Morgan Kauffman, San Francisco, California, 2001.
21. M. Pelikan, K. Sastry, and E. Cantú-Paz. *Scalable Optimization via Probabilistic Modeling*. Springer, Berlin Heidelberg New York, 2006.
22. W. F. Punch, D. Zongker, and E. D. Goodman. The royal tree problem, a benchmark for single and multiple population genetic programming. In P. J. Angeline and K. E. Kinnear, Jr., (eds.), *Advances in Genetic Programming 2*, pages 299–316. MIT, Cambridge, MA, USA, 1996.
23. J. Rissanen. Hypothesis selection and testing by the MDL principle. *The Computer Journal*, 42(4):260–269, 1999.
24. J. P. Rosca and D. H. Ballard. Discovery of subroutines in genetic programming. In P.J. Angeline and K. E. Kinnear, Jr., (eds.), *Advances in Genetic Programming 2*, chapter 9, pages 177–202, MIT, Cambridge, MA, 1996.
25. R. P. Salustowicz and J. Schmidhuber. Probabilistic incremental program evolution. *Evolutionary Computation*, 5(2):123–141, 1997.
26. K. Sastry and D. E. Goldberg. Probabilistic model building and competent genetic programming. In R. L. Riolo and B. Worzel, (eds.), *Genetic Programming Theory and Practise*, chapter 13, pages 205–220, Kluwer, Dordrecht 2003.
27. Y. Shan, R. I. McKay, R. Baxter, H. Abbass, D. Essam, and H. X. Nguyen. Grammar model-based program evolution. In *Proceedings of the 2004 Congress on Evolutionary Computation – CEC2004*, IEEE, Piscataway, New Jersey, 2004.
28. D. Thierens. Scalability problems of simple genetic algorithms. *Evolutionary Computation*, 7(4):331–352, 1999.
29. D. Thierens and D. E. Goldberg. Mixing in genetic algorithms. In S. Forrest, (ed.), *Proceedings of the fifth conference on Genetic Algorithms*, pages 38–45, Morgan Kaufmann, USA, 1993.