GraphEA: a Versatile Representation and Evolutionary Algorithm for Graphs

Eric Medvet[1] and Alberto Bartoli[1]

Department of Engineering and Architecture, University of Trieste, Trieste, IT

Abstract. Many practically relevant computing artifacts are forms of graphs, as, e.g., neural networks, mathematical expressions, finite automata. This great generality of the graph abstraction makes it desirable a way for searching in the space of graphs able to work effectively regardless of the graph form and application. In this paper, we propose GraphEA, a modular evolutionary algorithm (EA) for evolving graphs. GraphEA is modular in the sense that it can be tailored to any graph optimization task by providing components for specifying how to vary edges and nodes, and how to check the validity of a graph. We designed GraphEA by building on previous EAs dealing with particular kinds of graphs and included a speciation mechanism for the preservation of structural innovations and one for the gradual complexification of the solutions structure. To validate the generality of GraphEA, we applied it to 3 radically different tasks (regressions, in two flavors, text extraction from examples, evolution of controllers for soft robots) and compared its effectiveness against less general approaches. The results are promising and indicate some directions for further improvement of GraphEA.

Keywords: Optimization · Symbolic regression · DFAs · Neuroevolution

1 Introduction and related works

The ever more ubiquitous neural networks, mathematical expressions that model in an interpretable way the dependency of one variable on other variables, finite automata constituting a broad model for computation, are all actionable forms of *graphs*. Graphs are thus an abstraction of great practical relevance. It would be hence desirable to leverage this generality for enabling the usage of a general optimization technique that, given a way for evaluating the quality of a graph, searches the space of graphs for the one with greatest quality. On the one hand, that optimization technique should be applicable to *all* kinds of graphs, i.e., regardless of the nature of their nodes and edges, without requiring the user to tune many parameters; on the other hand, it should be effective enough to be useful in practice.

In this paper, we head towards the ambitious goal of designing that optimization technique by proposing an evolutionary algorithm (EA) and a representation for evolving graphs, that we called GraphEA. GraphEA works on any graph, directed or undirected, whose nodes are defined over a predefined set and edges are labeled with labels defined over another set. GraphEA is a form of evolutionary computation (EC). The optimization consists in evolving a fixed-size population of candidate solutions (here graphs) to a given problem by means of the reiterated applications of two steps: selection and variation. For variation in particular, GraphEA is modular: it provides a template scheme for variation of the solutions that has to be instantiated to a particular form of graphs by the user. For example, when using GraphEA for evolving neural networks, edges are labeled with real numbers (the weights of the corresponding connections) and one variation consists in perturbing those numbers with some noise.

While designing GraphEA, we took into account previous works in which EC has been applied to graphs and included in our EA (a) a mechanism for preserving the structural innovation introduced by variation operators (*speciation*) and (b) a mechanism for starting from an initial population of simple graphs that get more complex during the evolution (*complexification*).

We performed an experimental evaluation of GraphEA for validating the claim that it is general and sufficiently effective in the search. To this aim, we considered three radically different problems in which the graph abstraction is instantiated in different ways: *regression*, where we use graphs for representing networks of mathematical operators or of base univariate functions, *text extraction from examples*, where graphs represent discrete finite automata, and *neuroevolution of controllers for soft robots*, where graphs represent neural networks with free topology. We compared our proposed approach to challenging baseline methods specific to each problem and the results are highly promising.

The idea of applying EC for optimizing a graph is not new. Several applications have been proposed and some approaches exhibiting some degrees of generality exist, e.g., [18, 23]. However, only recently researchers devoted their effort to design a general approach that can be applied to any kind of graphs, as we do in this paper: one of the most interesting proposals is likely the one by Atkinson et al. [2, 1]. In their EGGP (Evolving Graphs by Graph Programming), the authors rely on rule-based graph programming to perform variation in an evolutionary optimization search. In a recent study [20], EGGP is experimentally compared against other representations and EAs suitable for evolving graphs: the authors conclude that there is not a single EA, nor a single representation, that systematically outperforms the other options in the considered problems. This result suggested that more research has to be done about general optimization techniques for graphs and hence somehow motivated us in designing GraphEA. For space constraints, we cannot present a direct comparison against EGGP: we reserve this activity for future work.

2 GraphEA

We consider the task of optimization in the set $\mathcal{G}_{\mathcal{N},\mathcal{E}}$ of *directed decorated graphs*, later simply graphs, defined over a set $\mathcal{N}$ of nodes and a set $\mathcal{E}$ of edge labels.

A graph $g \in \mathcal{G}$ is a pair (N, e) composed of a set $N \subseteq \mathcal{N}$ of nodes and a function $e : N \times N \to \mathcal{E} \cup \varnothing$. N is the set of nodes of g: for a given pair $n_1, n_2 \in$

$N \times N$, an edge connecting n_1 to n_2 exists in g if and only if $e(n_1, n_2) \neq \varnothing$ and its label is $e(n_1, n_2) \in \mathcal{E}$. Note that, in general, $e(n_1, n_2) \neq e(n_2, n_1)$.

We assume that, for the purpose of optimization, a *validity predicate* $v : \mathcal{G}_{\mathcal{N},\mathcal{E}} \to \{\textsc{True}, \textsc{False}\}$ is available for delimiting a subset $\mathcal{G}'_{\mathcal{N},\mathcal{E}} \subseteq \mathcal{G}_{\mathcal{N},\mathcal{E}}$ in which performing the search: we say that a graph g is *valid* iff $v(g)$ is true, i.e., iff v belongs to $\mathcal{G}'_{\mathcal{N},\mathcal{E}}$. Moreover, we assume that a *fitness function* $f : \mathcal{G}'_{\mathcal{N},\mathcal{E}} \to \mathbb{R}$ exists which measures the quality of any valid graph for a given task. The task consists in finding a graph that maximizes (or minimizes) f.

2.1 Representation

GraphEA employs a direct representation, i.e., genetic operators operate directly on graphs. However, the representation is general, not tight to a specific kind of nodes or edge labels, i.e., specific $\mathcal{N}$ or $\mathcal{E}$ sets. Consistently, the genetic operators are agnostic with respect to the nature of nodes and edges, but assume that ways for building or modifying nodes and edges are available: we call *factory* a stochastic method for obtaining a node or edge label in, respectively, $\mathcal{N}$ or $\mathcal{E}$; we call mutation a stochastic method for modifying an edge label, i.e., a stochastic operator from $\mathcal{E}$ to $\mathcal{E}$. We denote by $n \sim f_{\mathcal{N}}$ ($e \sim f_{\mathcal{E}}$) a node (edge label) obtained from a node (edge label) factory $f_{\mathcal{N}}$. We denote by $e \sim m_{\mathcal{E}}(e')$ an edge label obtained by mutating an edge label e' with a mutation $m_{\mathcal{E}}$.

In Section 3, we present three cases of application of GraphEA to different domains, i.e., with different sets $\mathcal{N}, \mathcal{E}$ along with the corresponding factories and mutations.

In the following, we describe the three mutation operators $\mathcal{G}_{\mathcal{N},\mathcal{E}} \to \mathcal{G}_{\mathcal{N},\mathcal{E}}$ used in GraphEA (edge addition, edge modification, node addition). For all the operators, if, after the application of the operator on the parent the resulting graph is not valid, then the parent graph is returned, hence ensuring the closure property with respect to $\mathcal{G}'_{\mathcal{N},\mathcal{E}}$.

Despite there are some indications in the EC literature that the crossover operator is beneficial in specific cases (e.g., [8, 6]), in a preliminary experimental evaluation we found that this operator does not significantly improve the optimization effectiveness and efficiency of GraphEA, while requiring a fair amount of additional complexity in the representation; we hence decided to not include it for the sake of simplicity.

Edge addition. Given an edge label factory $f_{\mathcal{E}}$, this operator builds the offspring graph $g_c = (N_c, e_c)$ from a parent graph $g_p = (N_p, e_p)$ as follows. First, N_c is set to N_p. Then, a random pair of nodes n_1, n_2 is chosen in N_c^2 such that $e_c(n_1, n_2) = \varnothing$, i.e., that they are not connected in the parent graph. Then e_c is set to behave as e_p with the exception of the input n_1, n_2 for which $e_c(n_1, n_2) = e \sim f_{\mathcal{E}}$.

In other words, the edge addition adds a new edge to the graph with an edge label obtained from $f_{\mathcal{E}}$.

Edge modification. Given an edge label mutation $m_\mathcal{E} : \mathcal{E} \to \mathcal{E}$ and a mutation rate $p_\text{mut} \in [0,1]$, this operator builds the offspring graph $g_c = (N_c, e_c)$ from a parent graph $g_p = (N_p, e_p)$ as follows. First, N_c is set to N_p. Then, for each pair of nodes $n_i, n_j \in N_c^2$ for which $e_p(n_i, n_j) \neq \varnothing$, $e_c(n_i, n_j)$ is set to $e_p(n_i, n_j)$ with probability $1 - p_\text{mut}$ or to $e \sim m_\mathcal{E}(e_p(n_i, n_j))$ with probability p_mut.

In other words, the edge modification modifies the labels of a fraction p_mut (on average) of existing edges using $m_\mathcal{E}$.

Node addition. Given two edge label mutations $m_\mathcal{E}^\text{src}, m_\mathcal{E}^\text{dst}$ and a node factory $f_\mathcal{N}$, this operator builds the offspring graph $g_c = (N_c, e_c)$ from a parent graph $g_p = (N_p, e_p)$ as follows. First, a pair of nodes n_1, n_2 is chosen in N_c^2 such that $e_p(n_1, n_2) \neq \varnothing$. Second, a new node $n \sim f_\mathcal{N}$ is obtained from the node factory $f_\mathcal{N}$. Third, N_c is set to $N_p \cup \{n\}$. Finally, e_c is set to behave as e_p with the exceptions of the three input pairs (n_1, n_2), (n_1, n), (n, n_2), for which it holds $e_c(n_1, n_2) = \varnothing$, $e_c(n_1, n) = e_\text{src} \sim m_\mathcal{E}^\text{src}(e_p(n_1, n_2))$, and $e_c(n, n_2) = e_\text{dst} \sim m_\mathcal{E}^\text{dst}(e_p(n, n_2))$.

In other words, the node addition selects an existing edge, removes it, and adds a new node obtained from $f_\mathcal{N}$ in the middle of the two endpoints of the removed edge: the endpoint are then connected to the new node with edges whose labels are mutations of the removed edge.

2.2 Evolutionary algorithm

Two of the three genetic operators described in the previous section can introduce structural modifications in a graph, i.e., the number of nodes of edges can change. In the context of the evolutionary optimization, those structural modifications are innovations that can be, on the long term, beneficial; yet, on the short term, they might affect negatively the fitness of a graph. In order to allow the structural modifications enough time to express their potential, if any, we employ in GraphEA an innovation protection mechanism.

In brief, the protection mechanism is a form of *speciation* based on fitness sharing [21] inspired by the similar mechanism employed in NEAT [23]. Individuals in the population are partitioned in species: all the individuals of the same species have the same chance of reproducing that depends on the fitness of one representative of the species; moreover, species larger than a predefined size have their best individuals moved in the next generation, as well as the global best (a form of *elitism*). Intuitively, an innovative individual resulting from a structural modification of a fit parent will likely belong to the same species of the parent and hence will not be penalized, in terms of chances of reproducing, if its fitness is lower. However, if it is fitter, the entire species will potentially benefit, having a higher chance to reproduce.

Beyond the speciation mechanism, the generational model of GraphEA is based on a fixed-size population that is updated iteratively without overlapping (i.e., the offspring replaces the parents) and with elitism. The offspring is built from the parents by applying, for each individual, one of the three mutation operators chosen with predefined probabilities. For space constraints, we give the detailed description of the EA of GraphEA in Appendix A.

The salient part of the EA of GraphEA is the speciation procedure: given a collection of graphs P and a threshold τ (a parameter of GraphEA) this procedure returns a partition $(P_1, P_2, \dots)$ of P. The partition is built iteratively in a bottom-up agglomerative fashion starting from the empty set of subsets of P, as follows. For each $g \in P$, if the partition is empty, a new subset $\{g\} = P_1$ is added to the partition; otherwise, the distances $d_1, d_2, \dots$ between g and the representative of each subset P_i are computed—the representative being the first graph added in each subset. Then the shortest distance $d_{i^\star}$, with $i^\star = \arg\min_i d_i$ is compared against the threshold τ: if $d_{i^\star} < \tau$, g is added to $P_{i^\star}$; otherwise a new subset $\{g\}$ is added to the partition. As distance between graphs we use the Jaccard distance between the corresponding sets of nodes.

3 Experimental evaluation

We performed three suites of experiments by applying GraphEA to three different domains: (symbolic) regression, text extraction, evolution of controllers of simulated soft robots.

The goals of the experiments were two: (a) demonstrate the general applicability of GraphEA to radically different domains and (b) verify that this generality does not come at the cost of unpractical search effectiveness. For the latter goal, we considered for each of the three cases at least one viable alternative based on a different representation, that we used as a baseline for the search effectiveness. We did not finely tune the main parameters of GraphEA: after a very shallow exploratory analysis to $\tau = 0.25$, $s_{\text{min}} = 5$ (see Appendix A), $\alpha = 0.75$ (see Appendix A); similarly, we set $p_{\text{edge-add}} = 0.6$, $p_{\text{edge-mod}} = 0.2$, and $p_{\text{node-add}} = 0.2$, unless otherwise indicated.

We performed all the experiments with a Java framework for evolutionary optimization (JGEA, publicly available at `https://github.com/ericmedvet/jgea`), that we augmented with a prototype implementation of GraphEA. For the experiments involving the simulated robots, we used 2D-VSR-Sim [16].

3.1 Regression

The goal of regression is to fit a model that accurately describes how an dependent numerical variable y depends on one or more independent numerical variables $\boldsymbol{x}$, based on a learning set of observations $\{\boldsymbol{x}_i, y_i\}_i$. If the space of the models is the space of mathematical expressions, this task is called *symbolic regression* (SR). Symbolic regression is one of the most crowded playfields in EC, the most prominent role being played by tree-based genetic programming (GP) [12]. Building on plain GP, many improvement are continuosly proposed and evaluated on SR (e.g., [25]). At the same time, the ability of GP to solve practical SR problems has been exploited more and more in other research fields [7].

We applied GraphEA to SR in two ways. In the first case, the graphs evolved by GraphEA are a generalization of the abstract syntax trees of the common tree-based GP: nodes are either mathematical operators, variables (input or output), or constants; directed edges are not labeled—we denote this variant by OG-GraphEA, OG standing for operator-graph. In the second case, the graphs are networks of univariate basic functions, each one processing a weighted sums of incoming edges; this representation basically corresponds to compositional pattern producing networks (CPPNs) [22] and we denote it by CPPN-GraphEA.

As fitness function for regression, we use the Mean Squared Error (MSE) with linear scaling, that have been showed to be beneficial when tackling symbolic regression with GP [11]. Let $\{\hat{y}_i\}_i$ be the output of a candidate solution on the input $\{\boldsymbol{x}_i\}_i$, its fitness is $\min_{a,b} \frac{1}{n} \sum_{i=1}^{i=n} (a\hat{y}_i + b - y_i)^2$, n being the number of input cases.

OG-GraphEA for SR. In this representation, graphs are directed and nodes are either independent variables ($x_1, x_2, \dots$, the actual number depending on the specific problem), the dependent variable y, constants (0.1, 1, and 10), and mathematical operators (we experimented with $+$, $-$, $\times$, p$\div$, and plog). p$\div$ and plog are the protected versions of the division and the logarithm, with x p$\div$ $0 \triangleq 0, \forall x$ and $\text{plog}(x) \triangleq 0, \forall x \leq 0$. Edges are not actually labeled in OG-GraphEA: formally $\mathcal{E}$ is a singleton with a single placeholder element e_0. Since, by definition, the set of nodes N in a graph cannot contain duplicates, for allowing a graph to contain many nodes with the same mathematical operator, we formally set $\mathcal{N}$ to $\{x_1, x_2, ...\} \cup \{y\} \cup \{0.1, 1, 10\} \cup \{+, -, \times, \text{p}\div, \text{plog}\} \times \mathbb{N}$: in other words, operator nodes are decorated with an index that does not matter when using the graph for computing an y value out of a $\boldsymbol{x}$.

The validity predicate for OG-GraphEA tests if a graph g meets all the following criteria: (i) g is acyclic, (ii) the node y has exactly 1 predecessor (i.e., another node n for which $e(n, y) = e_0$) and no successors, (iii) for each operator node, it has the proper number of predecessors (≥ 1 for $+$ and $\times$, 2 for $-$ and p$\div$, 1 for plog), (iv) for each constant node, it has no predecessors.

Concerning the genetic operators, the edge label factory used in the edge addition always returns e_0. The edge modification operator is disabled (i.e., $p_\text{edge-mod} = 0$), since it is not meaningful. In node addition, both edge label mutations are set to the identity (i.e., $m_\mathcal{E}^\text{src}(e_0) = m_\mathcal{E}^\text{dst}(e_0) = e_0$) and the node factory produces a randomly chosen mathematical operator indexed with a random integer (chosen in a sufficiently large range in order to rarely have two nodes with the same operator and index in the same graph).

Finally, concerning the initialization of the population, it builds valid graphs with all the variable and constant nodes and no operator nodes. As a direct consequence of the validity predicate, only one node (either a constant or a independent variable node) is connected to the output node y. This form of initialization resembles the complexification principle of NEAT: the evolution starts from simple solutions, hence benefiting from a smaller search space, and then makes them more complex, as needed, during the evolution.

CPPN-GraphEA for regression. In this representation, graphs are directed, nodes are either x_i, y, the constant 1, or base functions $\mathbb{R} \to \mathbb{R}$ (we experimented with the ReLu, the Guassian, $1 \text{ p}\!\div x$, and x^2). Edge labels are real numbers (i.e., $\mathcal{E} = \mathbb{R}$). As for OG-GraphEA, in $\mathcal{N}$ the base functions are decorated with an index.

The validity predicate for CPPN-GraphEA tests if a graph g meets all the following criteria: (i) g is acyclic, (ii) the node y has exactly at least one predecessor and no successors, (iii) for each constant or x_i node, it has no predecessors.

The edge label factory in the edge addition genetic operator is the Gaussian distribution with 0 mean and unit variance ($f_\mathcal{E} = N(0,1)$). The edge label mutation in the edge modification consists in perturbing the label with a Gaussian noise $N(0,1)$. The edge label mutations in the node addition are the identity, for the source node, and a replacement with a new label chosen with $N(0,1)$, for the target node; the node factory produces decorated base functions with uniform probability, as for OG-GraphEA.

For the population initialization, the same complexification principle of OG-GraphEA is followed. Graphs in the initial populations contains only x_i, y, and the constant and all the possible edges are present, labeled with a value chosen with $N(0,1)$.

Procedure and results. We considered a set of four "classical" symbolic regression problems, chosen according to the indications of [28]: Keijzer-6 ($\sum_{i=1}^{i=\lfloor x_1 \rfloor} \frac{1}{i}$), Nguyen-7 ($\log(x_1 + 1) + \log(x_1^2 + 1)$), Pagie-1 ($\frac{1}{1+x_1^{-4}} + \frac{1}{1+x_2^{-4}}$), and Poly4 ($x_1^4 + x_1^3 + x_1^2 + x_1$)—we refer the reader to [28] and our publicly available code for the details about the fitness cases for each problem.

As baselines, we considered a standard implementation of GP (with the same building blocks of OG-GraphEA) and a grammar-based version of GP (CFGGP [27], with a grammar specifying the same operators and constants of GP and OG-GraphEA) augmented with a diversity promotion strategy [3] that consists in attempting to avoid generating individuals whose genotype is already present in the population. For both GP and CFGGP we reproduce individuals using subtree-crossover for 0.8 of the offspring and subtree-mutation for the remaining 0.2; we used tournament selection with size 5 for reproduction and truncation selection for survival selection after merging the offspring with the parents at each generation (i.e., overlapping generational model); we built the initial population with the ramped half-and-half method [15]. For each of the four EAs, we set $n_\text{pop} = 100$ and stopped the evolution after 100 generations.

For each problem and each EA, we performed 20 independent runs with different random seeds. Table 1 summarizes the results in terms of the fitness of the best solution at the end of the evolution (median and standard deviation across the runs).

It can be seen that CPPN-GraphEA obtains the best fitness in 3 on 4 problems, while not performing well on Keijzer-6. On the other hand, OG-GraphEA struggles in all the problems, obtaining the last or second-last effectiveness. We interpret this finding as a consequence of the different expressiveness of the

Table 1. Best final fitness for regression (the lower, the better).

EA	Keijzer-6	Nguyen-7	Pagie-1	Poly4
CFGGP	0.001±0.004	0.028±0.046	16.571±15.305	0.499±3.042
CPPN-GraphEA	0.242±0.229	0.001±0.003	2.169± 8.080	0.320±0.925
OG-GraphEA	0.010±0.060	0.149±0.214	24.340±25.211	3.807±2.673
GP	0.002±0.002	0.040±0.067	22.642±11.709	0.410±2.334

two representations: where the required degree of composition between building blocks is larger, CPPN-GraphEA finds good solutions by approximation, whereas OG-GraphEA is not able to (within the 100 generations) build the required substructures in the graph. As a further confirmation of this interpretation, we looked at the size of the found solutions (number of nodes and edges for GraphEA, number of tree nodes in GP and CFGGP) and found that it was much lower for the former: it turns out, hence, that the complexification strategy is detrimental for OG-GraphEA.

3.2 Text extraction

Syntax-based extraction of text portions based on examples is a key step in many practical workflows related to information extraction. Different approaches have been proposed for solving this problem, based, e.g., on classical machine learning [17], deep learning [26], or EC [5, 14]. One of the most effective approaches, in which the outcome of the learning from the examples is a regular expression, is based on GP [4]: building blocks for the trees are regular expression constructs and constants (i.e., characters) and the regular expression is obtained by traversing the tree in a depth-first order.

Formally, the text extraction problem is defined as follows. An extractor is a function that takes in input a string s and outputs a (possibly empty) set S of substrings of s, each one identified by the start and end index in s. A problem of text extraction consists in, given a string s and a set S of substrings to be extracted from s, learning an extractor that, when applied to s, returns S. In practical settings, the learned extractor should also generalize beyond the examples represented by s, S and learn the underlying pattern of substrings of interest. This additional objective makes the task harder [5]: we here do not focus on the generalization ability and consider instead just the simpler version of the text extraction problem. The fitness of a candidate extractor is the char error rate measured on a pair s, S: each character is considered as a data point in a binary classification setting, i.e., it can be a positive, if it belongs to a string of S, or a negative, otherwise. The char error rate, measured on a pair s, S of an extractor extracting $\hat{S}$ from S, is the ratio of characters in s that are misclassified by the extractor, i.e., that "belong" to S but not to $\hat{S}$ or the opposite.

We here exploited the generality of GraphEA for exploring a radically different approach with respect to GP for regular expressions: we evolve deterministic finite automata (DFAs) in which transitions are set of characters. Indeed, the idea of evolving a DFA for binary strings have been already proposed in [14]. In terms of GraphEA, given a learning dataset s, S in which the positives characters form a set $\mathcal{A}$, $\mathcal{E}$ is the set of non-empty subsets of $\mathcal{A}$, i.e., $\mathcal{E} = \mathcal{P}(\mathcal{A}) \setminus \emptyset$, and $\mathcal{N} = \{\textsc{True}, \textsc{False}\} \times \mathbb{N}$, i.e., nodes can be accepting or not accepting and are decorated with an index (for the same reason of OG-GraphEA and CPPN-GraphEA). When applying the DFA to an input string, we consider the node with index 0 as the starting state.

The validity predicate for CPPN-GraphEA tests if a graph g (i) has exactly one node decorated with 0 and, (ii) for each node, the union of the labels (that are subsets of $\mathcal{A}$) of all outgoing edges is empty.

As edge label factory for the edge addition operator we use a random sampling of singletons in $\mathcal{P}(\mathcal{A})$. The edge label mutation for the edge mutation operator works as follows: given an edge label A, if $|A| = 1$, it adds a random element of $\mathcal{A}$ to A; otherwise, if $A = \mathcal{A}$, it removes a random element from A; otherwise, it applies one of the previous modifications with equal probability. For both the edge label mutations in the node addition operator, we use the identity; the node factory samples with uniform probability $\{\textsc{True}, \textsc{False}\} \times \mathbb{N}_0$, i.e., picks randomly among all possible nodes not decorated with 0.

For the population initialization, we apply again the complexification principle. Graph in the initial population are composed of only two nodes—one being decorated with 0, and hence being the starting state, the other being an accepting node—connected by an edge labeled with a randomly chosen singleton of $\mathcal{P}(\mathcal{A})$.

Procedure and results. We defined a procedure for building synthetic datasets s, S for text extraction based on two parameters related to the problem hardness: the size $n_s \leq 10$ of the alphabet of s and the number of positive examples n_S. Regardless of the values of these parameters, the set S of substrings is always composed of the matches of the following three regular expressions: `000000`, `111(00)?+(11)++`, and `(110110)++`. Given the values for n_s, n_S, a random s composed of the characters corresponding to the first n_s digits is generated randomly and incrementally until it contains at least n_S matches for each of the three regular expressions. We experimented with values $n_s = 2, n_S = 5$, $3, 5$, $4, 8$, and $4, 10$.

As a comparison baseline, we experimented with the same variant of CFGGP described in Section 3.1, with the same parameters and with a grammar tailored for building regular expressions composed of the characters in $\mathcal{A}$ and the constructs `(?:`r`)`, `.`, and r`|`r, r being a placeholder for another regular expression. Again, for both the EAs, we set $n_{\text{pop}} = 100$ and stopped the evolution after 100 generations.

For each problem and each EA, we performed 20 independent runs with different random seeds. Table 1 summarizes the results in terms of the fitness of the best solution at the end of the evolution (median and standard deviation across the runs).

Table 2. Best final fitness for text extraction (the lower, the better).

EA	$n_s = 2, n_S = 5$	$n_s = 3, n_S = 5$	$n_s = 4, n_S = 10$	$n_s = 4, n_S = 8$
CFGGP	0.160±0.030	0.120±0.050	0.092± 0.037	0.097± 0.044
GraphEA	0.169±0.027	0.093±0.035	0.045± 0.015	0.043± 0.008

The figures in Table 2 suggests that GraphEA outperforms CFGGP for all but the hardest problem ($n_s = 2, n_S = 5$), for which the difference is not neat. A representation based on graphs is perfectly suitable for learning DFAs (that are graphs) and DFAs are a natural choice for extractors, at least if a compact, human-readable representation of the extractor is required. As a result, GraphEA seems to be more capable of searching the space of solutions than CFGGP that, in this case, works on a less direct representation.

3.3 Robotic controller optimization

Robots composed of soft materials constitute a promising approach to many tasks for which rigid robots are not suitable. Due to their nature, they can interact with fragile or sensible objects or exhibit degrees of compliance to the environment that are not feasible with rigid counterparts. One category of soft robots that is of further interest is the one of voxel-based soft robots (VSRs), that are composed of many simple deformable blocks (the voxels) [10]. Beside being soft, VSRs are inherently modular and represent hence a stepping-stone towards the ambituous goal of auto-reproducing machines. Unfortunately, designing a VSRs is a complex task, because the nontrivial interactions among its many components are difficult to model. For this reason, VSR design has been tackled with optimization, often by means of EC [13, 24]: many aspects of a VSR can be subjected to optimization, most notably the shape and the controller. We here focus on the latter and consider the same scenario of [24], that we briefly summarize here—we refer the reader to the cited paper for the full details.

Given a 2-D shape for a VSR in which each one of the voxel is equipped with zero or more sensors (i.e., a VSR body), a controller is a law according to which the actuation value (a signal in $[-1, 1]$ corresponding to expansion or contraction of the voxel) of each voxel is determined at each time step based on the numerical values read by the sensors. Formally, the controller is hence a function from $\mathbb{R}^m$ to $\mathbb{R}^n$, m being the number of sensor readings and n the number of voxels. The problem of robotic controller optimization consists in learning the controller function that results in the best degree of achievement of a given task with a given body. We here focus on locomotion, and the fitness that measures the degree of achievement is the distance traveled by the VSR in a simulation lasting 20 s (simulated time).

For this problem, we used GraphEA for directly representing a artificial neural network (ANN) without a predefined topology, very similarly to NEAT—i.e., we realized a form of neuroevolution. All the parameters for the representation

($\mathcal{E}$, $\mathcal{N}$, the genetic operators, and the population initialization) are the same of CPPN-GraphEA, with the exceptions of the available base functions for the nodes, for which here we used only the tanh, and the output nodes, which are $y_1, \dots, y_n$.

Procedure and results We considered the task of locomotion and two bodies: a "worm" composed of 6×2 voxels and a "biped" composed of a 4×2 trunk and two 1×1 legs. In both cases, the voxels in the top row were equipped with sensors sensing the x- and y-rotated-velocity of the voxel (see [16]), the bottom row (i.e., the legs for the biped) voxels with touch sensors, and all the voxels with current area ratio sensors.

As a baseline, we used CMA-ES [9], a popular numerical optimization algorithm that proved to be particularly effective in many reinforcement learning settings, for optimizing the weights of a fixed-topology multilayer perceptron (MLP): following the findings of [24], we used a topology composed of a single hidden layer with $0.65(m+1)$ nodes, $m+1$ being the number of input nodes (one plus the bias). As an aside, the present experiment is, to the best of our knowledge, the first application of CMA-ES for optimizing the controller of a VSR; nevertheless, successful applications of CMA-ES has been proposed for other kinds of modular robots (e.g., [19]). We used the basic version of CMA-ES with an initial vector of means randomly extracted in $[-1, 1]^p$, p being the number of weights in the MLP. The two considered bodies corresponded to MLPs having $p \approx 400$ and $p \approx 800$ weights for the biped and the worm, respectively.

For each problem and each EA, we performed 20 independent runs with different random seeds. Since CMA-ES uses a population size that depends on the dimension p of the search space, we stopped the evolution, for both EAs, after 20 000 fitness evaluations, while still using $n_{\text{pop}} = 100$ for GraphEA. Table 1 summarizes the results in terms of the fitness of the best solution at the end of the evolution (median and standard deviation across the runs).

Table 3. Best final fitness for controller optimization (the greater, the better).

EA	Biped	Worm
GraphEA	65.0±8.3	43.5± 5.2
CMA-ES	94.7±9.9	104.0±10.4

It can be seen that in this case GraphEA does not compare favorably with the baseline, the gap in the final fitness being consistently large across all the experiments. In an attempt to understand this experimental observation, we analyzed the size (number of nodes and edges) of the graphs generated by GraphEA and found that, despite it consistently grows over the evolution, it never reaches the (fixed) size of the MLPs optimized by CMA-ES. Interestingly, the gap was larger for the worm (on average ≈ 320 vs. ≈ 445) than for the biped (on average ≈ 500

vs. ≈ 920) and the performance gap was larger for the worm than for the biped. We did not investigate in detail if every weight in the MLPs optimized by CMA-ES was actually important (ANNs may be pruned without being hampered in effectiveness), we think that the performance gap of GraphEA is at least partly due to the longest time it takes to evolve a graph that is complex enough for the task at hand. This limitation may be a consequence of many design choices for GraphEA, the most prominent being the initialization procedure following the complexification principle.

4 Concluding remarks

Our experimental evaluation of GraphEA on three radically different problems based on the graph abstraction shows that the proposed approach is indeed general and effective. Specifically, GraphEA is competitive with more specific forms of optimization tailored to regression and text extraction from examples, while it is clearly outperformed by a state-of-the-art technique in neuroevolution of soft robot controllers. While this outcome is in line with other recent findings [20], in the sense there is not a single EA, nor a single representation, that systematically outperforms the other options across different problems, by digging in the results we noticed that when GraphEA struggles in matching the effectiveness of other optimization techniques, it often produces solutions which are remarkably simpler (and hence likely less expressive) than those of the baseline counterpart. We interpret this finding as an opportunity to further improve GraphEA. We speculate that some form of self-tuning of the population initialization and variation operators, capable of adaptively driving the search to the exploration of the search space where solutions are more complex, could be beneficial not only to GraphEA, but also to similar approaches, as EGGP.

Acknowledgments. We thanks Luca Zanella for the CMA-ES implementation in JGEA. The experimental evaluation of this work has been done on CINECA HPC cluster within the CINECA-University of Trieste agreement.

References

[1] Atkinson, T.: Evolving graphs by graph programming. Ph.D. thesis, University of York (2019)
[2] Atkinson, T., Plump, D., Stepney, S.: Evolving graphs by graph programming. In: European Conference on Genetic Programming. pp. 35–51. Springer (2018)
[3] Bartoli, A., De Lorenzo, A., Medvet, E., Squillero, G.: Multi-level diversity promotion strategies for grammar-guided genetic programming. Applied Soft Computing 83, 105599 (2019)
[4] Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Can a machine replace humans in building regular expressions? a case study. IEEE Intelligent Systems 31(6), 15–21 (2016)

[5] Bartoli, A., De Lorenzo, A., Medvet, E., Tarlao, F.: Inference of regular expressions for text extraction from examples. IEEE Transactions on Knowledge and Data Engineering 28(5), 1217–1230 (2016)

[6] Corus, D., Oliveto, P.S.: Standard steady state genetic algorithms can hill-climb faster than mutation-only evolutionary algorithms. IEEE Transactions on Evolutionary Computation 22(5), 720–732 (2017)

[7] De Lorenzo, A., Bartoli, A., Castelli, M., Medvet, E., Xue, B.: Genetic programming in the twenty-first century: a bibliometric and content-based analysis from both sides of the fence. Genetic Programming and Evolvable Machines 21(1), 181–204 (2020)

[8] Doerr, B., Happ, E., Klein, C.: Crossover can provably be useful in evolutionary computation. Theoretical Computer Science 425, 17–33 (2012)

[9] Hansen, N.: The cma evolution strategy: a comparing review. In: Towards a new evolutionary computation, pp. 75–102. Springer (2006)

[10] Hiller, J., Lipson, H.: Automatic design and manufacture of soft robots. IEEE Transactions on Robotics 28(2), 457–466 (2012)

[11] Keijzer, M.: Scaled symbolic regression. Genetic Programming and Evolvable Machines 5(3), 259–269 (2004)

[12] Koza, J.R., Koza, J.R.: Genetic programming: on the programming of computers by means of natural selection, vol. 1. MIT press (1992)

[13] Kriegman, S., Blackiston, D., Levin, M., Bongard, J.: A scalable pipeline for designing reconfigurable organisms. Proceedings of the National Academy of Sciences (2020), `https://www.pnas.org/content/early/2020/01/07/1910837117`

[14] Lucas, S.M., Reynolds, T.J.: Learning deterministic finite automata with a smart state labeling evolutionary algorithm. IEEE transactions on pattern analysis and machine intelligence 27(7), 1063–1074 (2005)

[15] Luke, S.: Essentials of metaheuristics, vol. 113. Lulu Raleigh (2009)

[16] Medvet, E., Bartoli, A., De Lorenzo, A., Seriani, S.: 2d-vsr-sim: A simulation tool for the optimization of 2-d voxel-based soft robots. SoftwareX 12, 100573 (2020)

[17] Medvet, E., Bartoli, A., De Lorenzo, A., Tarlao, F.: Interactive example-based finding of text items. Expert Systems with Applications p. 113403 (2020)

[18] Miller, J.F., Harding, S.L.: Cartesian genetic programming. In: Proceedings of the 10th annual conference companion on Genetic and evolutionary computation. pp. 2701–2726 (2008)

[19] Miras, K., De Carlo, M., Akhatou, S., Eiben, A.: Evolving-controllers versus learning-controllers for morphologically evolvable robots. In: International Conference on the Applications of Evolutionary Computation (Part of EvoStar). pp. 86–99. Springer (2020)

[20] Sotto, L.F.D., Kaufmann, P., Atkinson, T., Kalkreuth, R., Basgalupp, M.P.: A study on graph representations for genetic programming. In: Proceedings of the 2020 Genetic and Evolutionary Computation Conference. pp. 931–939 (2020)

[21] Squillero, G., Tonda, A.: Divergence of character and premature convergence: A survey of methodologies for promoting diversity in evolutionary optimization. Information Sciences 329, 782–799 (2016)

[22] Stanley, K.O.: Compositional pattern producing networks: A novel abstraction of development. Genetic programming and evolvable machines 8(2), 131–162 (2007)

[23] Stanley, K.O., Miikkulainen, R.: Evolving neural networks through augmenting topologies. Evolutionary computation 10(2), 99–127 (2002)

[24] Talamini, J., Medvet, E., Bartoli, A., De Lorenzo, A.: Evolutionary Synthesis of Sensing Controllers for Voxel-based Soft Robots. In: The 2018 Conference on Artificial Life: A Hybrid of the European Conference on Artificial Life (ECAL) and the International Conference on the Synthesis and Simulation of Living Systems (ALIFE). pp. 574–581. MIT Press (2019)

[25] Virgolin, M., Alderliesten, T., Bosman, P.A.: Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In: Proceedings of the Genetic and Evolutionary Computation Conference. pp. 1084–1092 (2019)

[26] Wei, Q., Ji, Z., Li, Z., Du, J., Wang, J., Xu, J., Xiang, Y., Tiryaki, F., Wu, S., Zhang, Y., et al.: A study of deep learning approaches for medication and adverse drug event extraction from clinical text. Journal of the American Medical Informatics Association 27(1), 13–21 (2020)

[27] Whigham, P.: Inductive bias and genetic programming. In: First International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications. pp. 461–466. IET (1995)

[28] White, D.R., Mcdermott, J., Castelli, M., Manzoni, L., Goldman, B.W., Kronberger, G., Jaśkowski, W., O'Reilly, U.M., Luke, S.: Better gp benchmarks: community survey results and proposals. Genetic Programming and Evolvable Machines 14(1), 3–29 (2013)

A The algorithm of GraphEA

Algorithm 1 shows the iterative algorithm of GraphEA. The most salient part is in the offspring generation part (lines 4–20). After the population P is partitioned in n species (line 5, explained below), the offspring P' is composed by first adding the best individuals (lines 6–12) and then reproducing the individuals in each species (lines 13–20). In the first step, the overall best and the best of each species with size $\geq s_{\text{min}}$, a parameter of the algorithm, are added to P'.

```
1  function evolve():
2      $P \leftarrow$ initialize($n_{\text{pop}}$)
3      foreach $i \in \{1, \dots, n_{\text{gen}}\}$ do
4          $P' \leftarrow \emptyset$
5          $(\{g_{1,1}, \dots, g_{1,s_1}\}, \dots, \{g_{n,1}, \dots, g_{n,s_n}\}) \leftarrow$ speciate($P, \tau$)
6          $P' \leftarrow P' \cup \{$best($P$)$\}$
7          foreach $i \in \{1, \dots, n\}$ do
8              if $s_i \geq s_{min}$ then
9                  $P' \leftarrow P' \cup \{$best($\{g_{i,1}, \dots, g_{i,s_1}\}$)$\}$
10             end
11         end
12         $n'_{\text{pop}} \leftarrow n_{\text{pop}} - |P'|$
13         $\boldsymbol{r} \leftarrow$ ranks($g_{1,1}, \dots, g_{n,1}$)
14         foreach $i \in \{1, \dots, n\}$ do
15             $o \leftarrow n'_{\text{pop}} \alpha^{r_i} \frac{1}{\sum_{i=1}^{i=n} \alpha^{r_i}}$
16             foreach $c \in \{1, \dots, o\}$ do
17                 $P' \leftarrow P' \cup \{\text{mutate}(g_{i, c \bmod s_i})\}$
18             end
19         end
20         $P \leftarrow P'$
21     end
22     return best($P$)
23 end
```

Algorithm 1: The GraphEA algorithm.

In the second step, for generating the n'_{pop} offspring, one representative individual is randomly chosen in each of the n species: the representatives are then sorted according to their fitness and their rank is stored in $\boldsymbol{r}$. Then, a number o of offspring is reserved to each species depending on the rank of the corresponding representative, according to a rank-proportional scheme where $o = n'_{\text{pop}} \alpha^{r_i} \frac{1}{\sum_{i=1}^{i=n} \alpha^{r_i}}$. $\alpha \in]0, 1]$ is a parameter of the algorithm: the closer to 1, the less the preference to fittest species. Finally, the overall offspring for the next generation is completed by reproducing the individual of each species until a corresponding number o of new individuals are obtained. Since, in general, a species might get a reserved a number o larger than the current size, some individual of that species might reproduce more than one time ($c \bmod s_i$ in line 17).

Reproduction consists in the application of one of the three mutation operators presented above with probabilities $p_{\text{edge-add}}$, $p_{\text{edge-mod}}$, $p_{\text{node-add}}$ (summing to 1). The evolution terminates after n_{gen} iterations.