A General Purpose Representation and Adaptive EA for Evolving Graphs

Eric Medvet
DIA - University of Trieste
Trieste, Italy
emedvet@units.it

Simone Pozzi
DMG - University of Trieste
Trieste, Italy
simone.pozzi@studenti.units.it

Luca Manzoni
DMG - University of Trieste
Trieste, Italy
lmanzoni@units.it

ABSTRACT

Graphs are a way to describe complex entities and their relations that apply to many practically relevant domains. However, domains often differ not only in the properties of nodes and edges, but also in the constraints imposed to the overall structure. This makes hard to define a general representation and genetic operators for graphs that permit the evolutionary optimization over many domains. In this paper, we tackle this challenge. We first propose a representation template that can be customized by users for specific domains: the constraints and the genetic operators are given in Prolog, a declarative programming language for operating with logic. Then, we define an adaptive evolutionary algorithm that can work with a large number of genetic operators by modifying their usage probability during the evolution: in this way, we relieve the user from the burden of selecting in advance only operators that are "good enough". We experimentally evaluate our proposal on two radically different domains to demonstrate its applicability and effectiveness: symbolic regression with trees and text extraction with finite-state automata. The results are promising: our approach does not trade effectiveness for versatility and is not worse than other domain-tailored approaches.

CCS CONCEPTS

• **Computing methodologies** → **Genetic programming**; *Logic programming and answer set programming*.

KEYWORDS

Graph Evolutionary Algorithm, Declarative Programming, Prolog, Adaptive Evolutionary Algorithm

1 INTRODUCTION

Graphs are an (or even *the*) ubiquitous data structure, appearing in each aspect of computer science and in the modeling of countless kinds of phenomena. Examples range from social connections to the links between web pages, from dependencies in production pipelines to artificial neural networks. It would be hard to find a field in which graphs are not used in some modeling aspect.

As graphs are so important, finding a graph that optimizes some measure, i.e., graph optimization, is in turn a very relevant task. Due to the discrete nature of graphs, optimization techniques able to work without a gradient or to perform discrete modifications are preferable, thus leading to an evolutionary approach as one of the most natural. The universality of graphs, however, is also a limiting factor to applying such techniques. While graphs appear in practically any field, the actual family of relevant graphs and the corresponding constraints vary greatly. For example, a feed forward neural network can be a directed acyclic graph where each edge has a label (the weight of the connection) and so does each node (the activation function). In modeling a social network edges might be directed or not, labeled with the type of connection or unlabeled. Thus, every method able to generate—or evolve—graphs must not only provide a way of representing graphs in general, but also a way to define which specific families of graphs should be part of the search space.

The idea of an evolutionary algorithm (EA) able to evolve graphs is certainly not new: for example, Cartesian GP [10] is a well established method that has been widely applied. A particular and well known kind of graph evolution is represented by neuroevolution techniques, of which NEAT (Neuroevolution of Augmenting Topologies) [13] and its successors [11] provide one of the most successful examples. However, in all those cases the evolution is limited to the class of graphs for which the algorithm has been defined, and any extension relies on changing (part of) the algorithm. More recently, EGGP (Evolving Graphs by Graph Programming) [1, 2] has been defined to be applicable in a more general way by making use of rule-based programming. Another general approach to the evolution of graphs is given by GraphEA [7], where the class of graphs that are evolved respects a validity predicate and the evolutionary process preserve the validity of that predicate.

In this work we provide a more general EA where the task of defining a subclass of valid graphs and the operators that act on them are separated from the general inner working of the EA itself. This is accomplished in two ways: (a) by using a *declarative* language (Prolog) with enough expressive power to define the family of graphs to act upon and the (unary) genetic operators and (b) an adaptive mechanism to select the probability of applying each of the specified genetic operators. This allows a domain expert without a deep knowledge of EAs to define the class of graphs to act upon as a collection of conditions, not by the code checking them, and to specify the genetic operators without having to consider their

GECCO '23, July 15–19, 2023, Lisbon, Portugal

ACM ISBN 979-8-4007-0119-1/23/07…$15.00
https://doi.org/10.1145/3583131.3590431

effectiveness: the most effective ones will be given priority by the adaptive mechanism.

To show the wide applicability of the proposed approach, we compared with existing specialized techniques in two different domains: tree-based symbolic regression and text extraction with DFAs. Results show that our approach is competitive with existing techniques while being more general, thus showing that "universality" does not have to necessarily come at the expense of performances.

2 GOAL OF THE STUDY

We aim at facilitating the evolutionary optimization over set of graphs. In particular, we target the case where a user wants to find a graph that (a) is consistent with some user-specified domain and (b) minimizes (or maximizes) a user-defined fitness function. We want the user to be required to provide just a description of the domain of interest and the fitness function, relieving them from the burden of defining low level details of the evolutionary optimization process. However, we also want to permit the user to provide some convenient ways to modify the graph in the considered domain, such that the optimization process may exploit them for a more effective or efficient search.

In the next subsections we formally define (a) the kind of graphs we are concerned with and (b) the components the user is required to provide to specify an optimization problem over a set of graphs.

2.1 Definition of graph

We consider the set of directed decorated graphs, later simply graphs, that we define as follows.

Let $\mathcal{A}$ be the set of *attributes* and $\mathcal{V}$ the set of *values*. We denote by $d(a) = \mathcal{V}_a \subseteq \mathcal{V}$ the *domain* of the attribute a. We denote by $m : \mathcal{A} \to \mathcal{V} \cup \{\varnothing\}$, with $\varnothing \notin \mathcal{V}$, a *mapping* of attributes to values such that each $a \in A$ maps to a value $m(a) \in \mathcal{V}_a \cup \{\varnothing\}$; we say that a is *valued* in m if $m(a) \neq \varnothing$. We denote by $\mathcal{M}$ the set of all possible mappings.

We define as *graph* a triplet $g = (M_N, M_E, e)$, where $M_N \subseteq \mathcal{M}$ is a set of *nodes*, each one being a mapping, $M_E \subseteq \mathcal{M}$ is a set of *edges*, each one being a mapping, and $e : M_N \times M_N \to M_E \cup \{\varnothing\}$ is a surjective function that maps each pair of nodes $m_{N,1}, m_{N,2}$ to either an edge $m_E = e(m_{N,1}, m_{N,2})$ or $\varnothing$; we say that two nodes $m_{N,1}, m_{N,2}$ are linked by the edge $m_E = e(m_{N,1}, m_{N,2})$ if $m_E \neq \varnothing$. We denote by $\mathcal{G}$ the set of all possible graphs.

We define as *subset of graphs* a set $G_P \subseteq \mathcal{G}$ for which every graph satisfies all the predicates in a set P: we write $p \models g$ if a graph g satisfies a predicate $p : \mathcal{G} \to \{\mathrm{T}, \mathrm{F}\}$.

***Example* 1** (Graphs for an online social network)**.** As an example, consider the subset of graphs modeling a (simple) online social network. Nodes could be either users or posts; users would have a username and posts would have a content. Each post should be linked to exactly one user by an edge representing authorship; each user could be linked to zero or more other users by edges representing the "follow" relation.

Formally, it would be $A = A_N \cup A_E$, with $A_N = \{\texttt{nodeType}, \texttt{username}, \texttt{text}\}$, $A_E = \{\texttt{edgeType}\}$, $d(\texttt{nodeType}) = \{\texttt{user}, \texttt{post}\}$, and $d(\texttt{edgeType}) = \{\texttt{authoredBy}, \texttt{followedBy}\}$—the domains of `username` and `text` being the set of strings. There would be some predicates constraining the structure of online social network graphs. For example, the predicate for the authorship would be: $\forall m_N : m_N(\texttt{nodeType}) = \texttt{post} \implies \exists! m'_N : m'_N(\texttt{nodeType}) = \texttt{user} \wedge e(m_N, m'_N)(\texttt{edgeType}) = \texttt{authoredBy}$. The predicate stating that all and only user nodes should have a username would be: $\forall m_N : m_N(\texttt{nodeType}) = \texttt{post} \veebar m_N(\texttt{username}) \neq \varnothing$ (where $\veebar$ represents the exclusive disjunction). Similar predicates would hold for the other properties. ■

2.2 User-provided specification of an optimization problem

We define an optimization problem over a subset $G \subseteq \mathcal{G}$ of graphs as a pair G, f, with $f : G \to \mathbb{R}$ being the fitness function. We here assume that the goal is to minimize[1] f, hence the problem can be described with $\arg\min_{g \in G} f(g)$.

In order to let the user specify an optimization problem G, f, we require them to provide the following components: (a) a set of predicates $P = \{p_i\}_i$, where each p_i is a predicate over $\mathcal{G}$, i.e., $p_i : \mathcal{G} \to \{\mathrm{T}, \mathrm{F}\}$, and such that P defines G, i.e., $G = G_P = \{g \in \mathcal{G} : \forall p \in P, p \models g\}$; (b) a set of operators $O = \{o_i\}_i$, where each o_i is a unary operator in G, i.e., $o_i : G' \to G$, with $G' \subseteq G$; (c) one initial graph $g_0 \in G$; (d) the fitness function $f : G \to \mathbb{R}$.

With the exception of the fitness function f, which is an obvious requirement for defining an optimization problem, the other three components deal with non trivial concepts related to graphs in general, as well as with the specific domain of the problem at hand. In this paper, we propose to use a single formalism to let the user specify all those three components in a domain-agnostic way. This formalism is based on Prolog, a well-established language for operating with logic. In the next section, we provide the necessary background on Prolog.

3 BACKGROUND: PROLOG

Prolog is a logic and declarative programming language developed in 1972. We used SWI-Prolog [16], which is a versatile implementation of the Prolog language. In particular, it provides a large set of built-in predicates that is further expandable by importing external modules.

3.1 Prolog syntax and entities

3.1.1 Terms and predicates. The single data type of Prolog is the *term*. Terms can be atoms, numbers, strings, variables, lists, or compound terms.

An *atom* is a sequence of letters, digits, and the underscore. An atom must start with a lowercase letter. For example, valid Prolog atoms are `atom`, `a`, `aTOM`, `a_12`.

A *number* is a sequence of digits, possibly preceded by a minus, possibly including one decimal separator. Numbers can hence represent integers or floats.

A *string* is a sequence of characters enclosed by double quotes.

[1] Since our work deals with the representation of the solutions, in the broad interpretation that also includes the genetic operators, and is agnostic with respect to the selection criteria and the generational model, it is portable to the larger class of problems of multi-objective optimization where at least a partial order relation is defined among solutions. For ease of reading, however, we describe our proposal in the context of single-objective optimization problems with total order.

A *variable* is a sequence of letters, digits, and the underscore. A variable must start with an uppercase letter or an underscore. For example `X`, `Variable`, and `_12` are all valid variables.

A *list* is composed of zero or more terms, in the form $[t_1, t_2, \dots]$, where each t_i is an element of the list, a term.

A *compound term* is composed of a functor and one or more arguments, in the form $f(t_1, t_2, \dots)$, where f is the functor, an atom, and each t_i is an argument, a term.

A *predicate* is a compound term or an atom. A *predicate indicator* is the composition of a predicate head and its arity in the form h/n, where h is the predicate head and n is the number of arguments (which can be 0).

3.1.2 Clauses and queries. Prolog is designed to describe relations defined by means of *clauses*. Clauses can be either facts or rules.

A *rule* relates one predicate with one or more other predicates, in the form h `:-` b., which reads "h is true if b is true". The left-hand side h of the rule is called head. The right-hand side b of the rule is called body: the body is a sequence of predicates separated by logic operators. In particular, the logic operator $\wedge$ (conjunction) is represented in Prolog by means of `,` (comma), while $\vee$ (disjunction) is `;` (semicolon). The definition of any clause must always ends up with a full stop.

A *fact* is a rule with an empty body, in the form h., and can be read as "h is true".

A *query* is a sequence of predicates separated by disjunctions or conjuctions, preceded by `?-`, and terminated by `.`, in the form `?-` $p_1 \circ p_2 \circ \dots$, where p_i are the predicates and each $\circ$ is either `,` or `;`.

3.1.3 Built-in predicates. SWI-Prolog is distributed with a large list of built-in predicates that can be used for several purposes, as `length/2` to check the size of a list, `integer/1` to verify if the given argument is an integer or not, or `true/0` and `false/0` always returning true and false. Moreover, it allows to import some modules providing other useful predicates, as the module `random` with predicates to generate random numbers. Other useful built-in predicates are the ones for comparison of terms, as `==/2` (used in the form t_1`==`t_2), which is true if the terms $t1$ and t_2 are equivalent, or its opposite `\==/2`, and the arithmetic predicates, as `>=/2`.

Built-in predicates also include a special kind of predicates called meta-predicates: meta-predicates, as detailed in the next section, may result in the modification of the state of execution. The two most important meta-predicates are `assert/1` and `retract/1` which are used to add or remove terms and clauses from the state.

3.2 Prolog execution

For the purpose of this paper, we consider the execution of a Prolog program, i.e., a sequence of clauses or queries, as the evolution of a dynamical system that starts from an initial state and is fed with the clauses of the program. We call *engine* that dynamical system. At each input, depending on the input and the state, the engine possibly updates the state and possibly produces an output.

The state of the engine is a set of clauses. The initial state is the set composed of the built-in predicates.

If the engine receives a clause as input, it adds it to the state and gives no output. Otherwise, if the engine receives a query as input, it attempts to find a *resolution* of the query and gives as output the outcome of the process. The output of a query is a pair composed of one Boolean value and zero or more variable mappings. The Boolean value is true if and only if the query—namely, the logical expression resulting from the conjunctions and disjunctions of its composing predicates—is a logical consequence of the current state, i.e., if it follows from the clauses in the state for at least one assignment of the variables in the query, if any. Each variable mapping is a tuple that maps the variables contained in the query predicate with values, such that the tuple makes the query true. If a query contains one or more meta-predicates, and if it is evaluated to true, all the changes made using those meta-predicates are reflected into the state. Otherwise, with no meta-predicates in the query, the engine does not update the state.

***Example* 2** (A Prolog execution)**.** Consider the following sequence of clauses and queries (we will omit trailing `.` for readability in inline Prolog snippets): (1) `animal(X) :- dog(X) ; elephant(X)`; (2) `dog(simba)`; (3) `elephant(dumbo)`; (4) `?- animal(simba)`; (5) `?- animal(X)`; (6) `?- dog(X)`; (7) `?- dog(X) , elephant(X)`; (8) `?- dog(X) ; elephant(X)`; (9) `?- dog(X) , elephant(Y)`.

Items (1) to (3) are clauses; namely, item (1) is a rule with a body and items (2) and (3) are facts. Item (1) says that "dogs and elephants are animals"; the two facts state that there are two named animals. After the engine consumes these three clauses, the state consists of the set containing them and the built-in predicates.

Items (4) to (9) are queries: all of them, with the exception of query of item (7), return true. The first query does not return any mapping, since it has no variables; the others return one or more mappings. `dog(X)` returns $\{(\texttt{X} \mapsto \texttt{simba})\}$; `dog(X) ; elephant(X)` returns $\{(\texttt{X} \mapsto \texttt{simba}), (\texttt{X} \mapsto \texttt{dumbo})\}$; `dog(X) , elephant(Y)` returns $\{(\texttt{X} \mapsto \texttt{simba}, \texttt{Y} \mapsto \texttt{dumbo})\}$. ■

4 PROLOG-BASED REPRESENTATION AND EA

We here describe how we let the user provide the components described in Section 2.2 for specifying an optimization problem on graphs. We also describe how these components are used to (a) transform a sequence of clauses into a graph and to (b) transform a sequence of clauses corresponding to a graph into another sequence of clauses corresponding to a potentially different graph. More formally, hence, we define and describe (a) a genotype-phenotype mapping function $\phi : \mathcal{P}^* \to \mathcal{G} \cup \{\varnothing\}$, where $\mathcal{P}$ is the set of valid Prolog clauses, $\mathcal{P}^*$ is the set of sequences of clauses (i.e., the *genotype* space or search space), and $\mathcal{G}$ is the set of graphs (i.e., the *phenotype* or solution space), and (b) a (template for) genetic operators $\varphi : \mathcal{P}^* \to \mathcal{P}^*$. We assume that ϕ may fail when mapping a C_g, not returning a valid graph: in this case, we set the output to $\varnothing$.

Finally, we describe an EA that exploits these genotype-phenotype mapping function and genetic operators, i.e., this representation.

4.1 Representation

4.1.1 User-provided components. As specified in Section 2.2, we require the user to provide only three components (besides the fitness function). In practice, in terms of our Prolog-based representation, we require: (a) a sequence of clauses C_G describing the predicates P

that define a given subset G of graphs; (b) a sequence of clauses C_{g_0} describing one graph $g_0 \in G$; (c) a set $Q = \{q_1, q_2, \dots\}$ of queries, each describing one operator $o : G' \to G$, with $G' \subseteq G$.

For all the three components, we leave the freedom to the user to express whatever facts or rules the Prolog syntax permits. We only require to adhere to the following conventions: (a) nodes have an unique id[2] and are defined with the fact `node_id(`i_N`)`, with i_N being the node id; (b) edges have an unique id and are defined with the fact `edge(`$i_{N,1}$`,`$i_{N,2}$`,`i_E`)`, with $i_{N,1}, i_{N,2}$ being the source and destination node ids and i_E being the edge id; (c) attributes values are defined with the fact a`(`i`,`v`)`, with a being the attribute, i the node or edge id, and v the value. That is, we require the user to use the predicates `node_id/1`, `edge/3`, and one a`/2` for each relevant attribute.

Concerning C_G, we require that it contains at least the definition of the domains of the attributes. For each attribute a, we require it to be defined with `attribute(`a`)` and its domain to be defined with one fact a`_val(`v`)` for each possible value v, if the domain is discrete, or with the rule a`_val(`V`) :- string(V)`, if the domain is the one of the strings (or using `float/1` or `integer/1`, instead of `string/1` for numbers). We also allow to specify limited numerical domains—we do not show here the corresponding rules for brevity. We require the user to define a rule whose head is `is_valid` and whose body is the conjunction of all the constraints on the graphs of the domain, i.e., whose body represents P.

Concerning C_{g_0}, we require it to simply contains the list of facts that correspond to the declaration of all the nodes, edges, and attribute mappings defining the graph g_0, using the predicates `node_id/1`, `edge/3`, and one a`/2` for attribute, as mentioned above.

Finally, we require that each $q \in Q$ is a Prolog query containing at least one meta-predicate and the predicate `is_valid/0`.

***Example* 3** (Components for the case of the online social network of Example 1)**.** The domain of the attribute `nodeType` would be defined with: (1) `nodeType_val(user)`; (2) `nodeType_val(post)`; Similarly for the attribute `edgeType`.

The attribute `username` would be defined with: (3) `attribute(username)`; (4) `username_val(X) :- string(X)`; (5) `username_val(null)`; where `null` corresponds to ∅, i.e., the value `username` takes for nodes of types `post`. Similarly for the attribute `text`.

The constraint on the authorship would be defined with: (6) one `Author(P) :- findall(A, (edge(P, A, X), edgeType(X, authoredBy)), Authors), length(Authors, L), L == 1`; (7) `authorship :- foreach(findall(P, nodeType(P, post), Posts), maplist(oneAuthor, Posts))`; where the first rule defines the predicate `oneAuthor/1` that is true for a post if it is properly linked to one single user and the second rule defines the predicate `authorship/0` that is true if the former is true for all the posts. All those facts and rules (including the others not shown here for brevity and the final `is_valid`) would constitute C_G.

Concerning C_{g_0}, a simple graph g_0 with one user that wrote one post and is followed by another user would correspond to the following C_{g_0}: (1) `node_id(user1)`; (2) `node_id(user2)`; (3) `node_id(post1)`; (4) `edge(user1, user2, edge1)`; (5) `edge(post1, user1, edge1)`; (6) `nodeType(user1, user)`; (7) `edge Type(edge1, followedBy)`; (8) and alike.

Concerning Q, we present here for brevity only two operators that can be applied to this example:

(1) `gensym(user, U), assert(node_id(U)), assert(nodeType(U, user)), assert(username(U, "name")), assert(text(U, null)), is_valid.`
(2) `findall((S, T, E), (edge(S, T, E), edgeType(E, followedBy)), Edges), random_member((U1, U2, ID), Edges), retract(edge(U1, U2, ID)), assert(edge(U2, U1, ID)), is_valid.`

The first one adds a new user to the graph; the second swaps the users linked by a `followedBy` edge. ■

4.1.2 From Prolog to graphs. For mapping a genotype, i.e., a sequence of clauses C_g, to a phenotype, i.e., a graph $g = \phi(C_g; C_G)$, given the domain C_G, we proceed as follows.

We start the Prolog engine (with default initial state, see Section 3.2) and we feed it with the concatenation $C_G \oplus C_g$ of the clauses C_G and C_g. Then, we input the engine with the query `?- is_valid`: if the output is false, then the current state of the Prolog engine does not correspond to a valid graph, hence we set $\phi(C_g; C_G)$ to ∅. Otherwise, if the output is true, we input the engine with a number of queries that result in the enumeration of all nodes, all edges, and all the attributes for all the nodes and edges. In this way, by consuming the output of these queries, we build the graph g.

In detail, we first consume the output of the query `?- node_id(N)`, obtaining the list of all the node ids as mappings of the variable `N`. Then, we consume the output of the query `?- edge(N1, N2, E)`, obtaining the list of all the edges as a mappings of the triplet of variables `N1`, `N2`, `E`, respectively the ids of the linked nodes and of the edge. We then consume the output of the query `?- attribute(A)` obtaining the list of all the attributes as mappings of the variable `A`. Finally, for each attribute a we consume the output of the query `?-` a`(V, I)` obtaining the list of all the attribute values as a mappings of the pair of variables `V`, `I`, respectively the value of the attribute and the id of the node or the edge.

We remark that the resulting g meets, if not ∅, the requirements defined by C_G "by design", and is hence an element of the subset $G \subseteq \mathcal{G}$ of graphs corresponding to the user-specified domain.

4.1.3 Application of a genetic operator. For applying a genetic operator represented by a query q to a genotype C_g, given the domain C_G, we proceed as follows.

We start the Prolog engine and we feed it with $C_G \oplus C_g$. Then, we input the engine with the query q which, we recall, contains the predicate `is_valid`. Then, we proceed as described in the previous section, obtaining either graph g' or ∅, if the query resulted in a false output. In the latter case, we set $\varphi(C_g; q, C_G)$ to C_g, i.e., we make the operator application not actually effective. Otherwise, we transform g' to a sequence of clauses $C_{g'}$ by setting one fact for each node (using `node_id/1`), one for each edge (using `edge/3`), and one for each attribute value different than ∅ (using a`/2`). Finally, we set $\varphi(C_g; q, C_G)$ to $C_{g'}$.

In practice, hence, when we apply φ, we modify the state of the Prolog engine with q, obtain the graph g' and map it back to a sequence of Prolog clauses (that we do not need to feed back to the

[2]Note that the unique id is not an actually additional requirement, since nodes and edges in practice have to be different and using an id is a practical and reasonable way to concretely realize their inequality.

Prolog engine). Note that, in principle, g' may result the same of g and hence $\varphi(C_g; q, C_G)$ may be C_g itself.

We remark that, whatever the outcome of the application operator, $\phi(\varphi(C_g; q, C_G), C_G) \in G$: that is, G is closed under the application of genetic operators.

4.2 An adaptive EA

We employ a rather simple mutation-only EA with overlapping. The EA is adaptive in the probability assigned to genetic operators, which is in turn used to select a genetic operator whenever an individual of the offspring has to be generated.

In detail, given the user-provided components C_G (corresponding to P and defining the domain $G \subseteq \mathcal{G}$), C_{g_0} (corresponding to g_0), $Q = \{q_1, q_2, \dots\}$ (corresponding to genetic operators O), and f (the fitness function), we proceed as follows.

Initially, we initialize a population C of n_{pop} genotypes starting from C_{g_0}. We build C by mimicking the ramped half-and-half procedure in such a way that the sizes of the resulting graphs are approximately evenly distributed in $[n_{\text{initMinSize}}, n_{\text{initMaxSize}}] \subset \mathbb{N}$, with the size of the graph being the sum of the number of nodes and edges and $n_{\text{initMinSize}}, n_{\text{initMaxSize}}$ being a pair of user-provided parameters of the EA. For this purpose, for each n in the size interval, we attempt to build $\left\lfloor \frac{1}{n_{\text{initMaxSize}} - n_{\text{initMinSize}} + 1} n_{\text{pop}} \right\rfloor$ individuals by repeatedly applying operators $q \in Q$ until the resulting graph has size n. We limit the maximum number of attempts to n_{attempts} in order to avoid endless execution when some graph sizes cannot be generated.

In the initialization phase, we also associate each operator $q \in Q$ with a weight $w(q) := 1$.

After the population initialization, we repeat for n_{gen} times, i.e., for n_{gen} generations, the following two steps. First, starting from the current population C, we generate the offspring C' of n_{pop} individuals by, for n_{pop} times, (i) selecting a parent genotype $C_g \in C$ using tournament selection with size n_{tour}, (ii) selecting an operator $q \in Q$ with probability proportional to the operator weight, and (iii) applying φ for obtaining the child genotype $C_{g'} = \varphi(C_g; q, C_G)$. In order to avoid the premature convergence resulting from the loss of diversity in the population [12], we repeat the three steps above until $\varphi(C_g; q, C_G) \neq C_g$, for at most n_{attempts} times—in this way we softly enforce diversity [3]. Once the offspring C' is built, we merge C and C' and retain only the n_{pop} best genotypes. While applying the operators, we keep track of the number $s(q)$ of times each operator q resulted in a child that was strictly better then the parent, i.e., such that $f(\phi(\varphi(C_g; q, C_G); C_G)) < f(\phi(C_g; C_G))$—$s(q)$ is hence a measure of success of q with respect to the fitness function f, i.e., it resembles a measure of evolvability [6, 8].

Second, we update the weights associated with the operators in Q based on their measured success $s(q)$, as follows. We sort the operators based on their decreasing $s(q)$, we increase each weight $w(q)$ of the top ρ% of the ranking to $w(q)(1+\eta)$, and we decrease each weight $w(q)$ of the bottom ρ% to $w(q)(1-\eta)$. We reset each $s(q)$ to 0 at each generation.

Upon the last generation, we take the graph with the best fitness as the result of the optimization.

Summarizing, our adaptive EA has the following domain-agnostic parameters: the population size n_{pop}, the number of generations n_{gen}, the tournament size n_{tour}, the maximum number of attempt n_{attempts}, the operator ranking proportion $\rho \in]0, 0.5]$, and the adaptation rate $\eta \in [0, 1]$. Moreover, the EA also uses the parameters $n_{\text{initMinSize}}, n_{\text{initMaxSize}}$, representing the size range of initial graphs, which are formally domain-independent, but that we tailor to the specific domains in the experimental evaluation described in the next sections.

5 EXPERIMENTAL EVALUATION

We performed a thorough experimental evaluation of our approach in order to (a) validate our claim that it is applicable to radically different domains and to (b) show that its broad applicability does not come at the cost of the search effectiveness. To this aim, we considered two different domains, applied our approach to them, and compared its results to those of domain-specific optimization techniques (one for each domain, constituting a baseline). For each domain, we customized only C_G, C_{g_0}, and Q, and left the remaining parts of our approach the same. That is, we pretended to be the user of the system who is required to provide the three domain-specific components in the form of Prolog clauses and queries.

Since one peculiarity of our approach is the EA being adaptive, we also conducted a further experimental analysis of the two corresponding parameters: the operator ranking proportion ρ and the schedule for the adaptation rate η. For the same reason, besides considering a baseline for each domain, we also considered three variants for our approach: the one described above (with adaptation and a "large" set Q of genetic operators), one with no adaptation and Q, one with no adaptation a smaller $Q_{\text{small}} \subseteq Q$.

We implemented our approach in JGEA [9], a Java-based evolutionary framework. For working with Prolog, we relied on JPL (https://jpl7.org/), a bidirectional interface between Java and Prolog. We made the code for our experiments publicly available[3]. Unless otherwise specified, we used the following values for the parameters: $n_{\text{pop}} = 70$, $n_{\text{gen}} = 100$, $n_{\text{tour}} = 5$, $n_{\text{attempts}} = 150$, $\rho = \frac{1}{3}$, $\eta = 0.01$. For each domain and each optimization technique, we performed 30 independent evolutionary runs by varying the random seed. When comparing the techniques, we carried out the Mann Whitney U rank test (after having verified the adequate hypotheses) with a significance level of $\alpha = 0.05$ (with Bonferroni correction).

5.1 Domain 1: tree-based symbolic regression

Symbolic regression is a popular domain in the field of evolutionary computation. Formally, the task consists in finding a mathematical expression that fits a dataset $\{\boldsymbol{x}_i, y_i\}_{i=1}^{i=n}$, with $\boldsymbol{x} \in \mathbb{R}^p$ and $y_i \in \mathbb{R}$.

We here considered 5 classic instances of symbolic regression which have been widely used as benchmark problems [15], three of them with $p = 1$, one with $p = 2$, and one with $p = 5$: (a) Keijzer-6, where $y = \sum_{j=1}^{j=\lfloor x_1 \rfloor} \frac{1}{j}$ and the dataset contains $n = 50$ points with x_1 evenly spaced in $[1, 50]$; (b) Nguyen-7, where $y = \ln(x_1+1)+\ln(x_1^2+1)$ and the dataset contains $n = 20$ points with x_1 randomly distributed in $[0, 2]$; (c) Pagie-1, where $y = \frac{1}{1+x_1^{-4}} + \frac{1}{1+x_2^{-4}}$ and the dataset contains $n = 625$ points with both x_1 and x_2 evenly spaced in $[-5, 5]$;

[3] https://github.com/SPozz/jgea/tree/features-graphs/it.units.malelab.jgea.sample/src/main/java/it/units/malelab/jgea/sample/lab/prolog

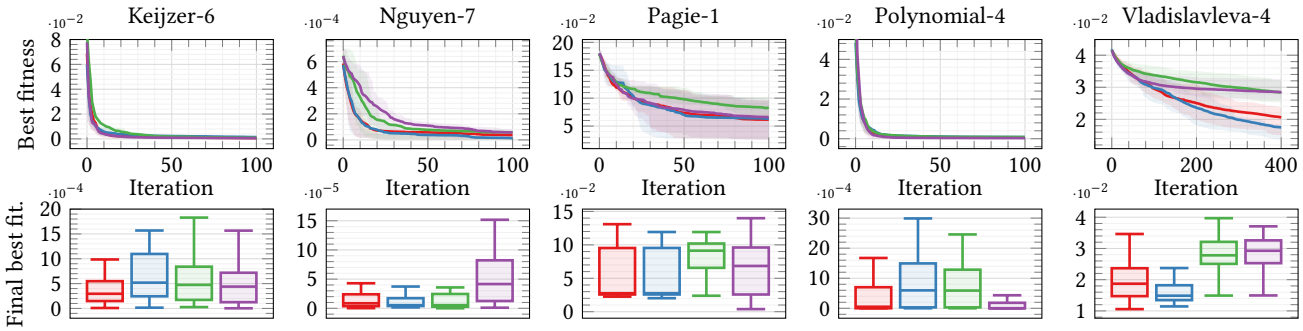

Figure 1: Results for the symbolic regression domain. Above: best fitness during the evolution (mean and interquartile range across 30 runs). Below: distribution of the final best fitness. Our approach with adaptation and Q is ▭, without adaptation and Q is ▭, without adaptation and Q_{small} is ▭; the baseline is ▭.

(d) Polynomial-4, where $y = x_1^4 + x_1^3 + x_1^2 + x_1$ and the dataset contains $n = 20$ points with x_1 evenly spaced in $[-1, 1]$; (e) Vlasdislavleva-4, where $y = \frac{10}{5+\sum_{i=1}^{5}(x_i-3)^2}$ and the dataset contains $n = 1024$ points with all variables randomly distributed in $[-0.05, 6.05]$. For Vlasdislavleva-4, being an harder and larger problem than the others, we set $n_{\text{gen}} = 400$, instead of 100.

We used the mean square error as fitness function; moreover, for both the baseline and our approach variants, we employed linear scaling while computing the error, as it has been shown to be a good practice in symbolic regression [14].

5.1.1 Baseline. One common way of solving symbolic regression problems is using tree-based GP [5], where mathematical expressions are represented as trees. We used tree-based GP as the baseline for this domain: in particular, we used a standard version of GP with overlapping, ramped half-and-half initialization, standard tree crossover (used for building 80 % of the offspring) and mutation (for the remaining 20 %) operators, tournament selection, and the same mechanism for the soft enforcement of diversity employed in our approach. We used standard operators as non-terminal nodes (i.e., $+$, $-$, $\cdot$, $\div$, log, with the protected versions of $\div$ and log) and the problem-specific variables and the constants $\{0.1, 1, 10\}$ as terminal nodes. For the common parameters (n_{pop}, n_{gen}, n_{tour}, n_{attempts}), we used the same values of our approach.

5.1.2 Customization of our approach. We set C_G to describe the subset of graphs corresponding to almost the same set of trees which can be expressed by the GP baseline, with the exceptions of the log operator, which we did not include here, and of the leaf nodes representing numerical constants, for which we permitted every value in $[0.001, 2]$.

Concerning $C_{g,0}$, we set it to describe the tree corresponding to the formula $x_1 + 1$, i.e., a very simple tree with 3 nodes.

Finally, concerning Q_{small} and Q, we set them as follows. In Q_{small} we put: (1) one query that replaces a leaf node with a subtree with a random operator and two random leaves (constants or variables); (2) one that replaces a terminal subtree of three nodes with a random constant or variable; (3) one that replaces a constant node value with a new one; (4) one that replaces an operator node with a new operator node. In Q we inserted all the items of Q_{small} and: (5) one query that perturbs a constant node value with a random rate in $]-0.25, 0.25[$; (6) as before, but in $]-0.1, 0.1[$; (7) one that adds the current tree as the first child of a new operator and adds a constant node as sibling; (8) one that behaves as the previous one but at a random non-root operator; (9) one that swaps two sibling leaves.

For the size range of the initial graphs, we used $[5, 125]$.

5.1.3 Results. Figure 1 shows the results of the experiments in symbolic regression. Namely, in the top row of plots, it shows how the fitness of the best individual in the population (mean and interquartile range across the 30 runs, as shaded area) varies during the evolution with the 4 methods (line color), on the five problem instances (plot). In the bottom row, it shows the distribution (in the form of boxplots) of the final best fitness.

The main finding is that our approach is never worse than the baseline: the versatility is hence not detrimental in terms of search effectiveness.

Interestingly, our approach in the main variant is significantly better ($p < 0.001$) than the baseline in the hardest problem, Vladislavleva-4. The same holds for the no-adaptation variant with the full Q, but does not hold for the variant with Q_{small}. This confirms that the possibility of adding several genetic operators by simply providing the corresponding Prolog queries, a form of exploitation of the knowledge of the domain, can be practically beneficial.

Finally, by comparing the adaptive version of our approach with the non-adaptive variants, we can see that the former is not worse than the latter ones in most problems, slightly better in two problems, slightly worse in one (the differences are never statistically significant, though).

5.2 Domain 2: text extraction with DFAs

In this domain, the task is to find an extractor of text that matches some given substrings of a given string. Formally, given a string s and a set S of substrings of s to be extracted, the extractor should match all and only the substrings in S.

This problem has been successfully tackled and solved with GP when the extractor has the form of a regular expression, internally represented as a tree [4]. Nevertheless, here, to show the versatility

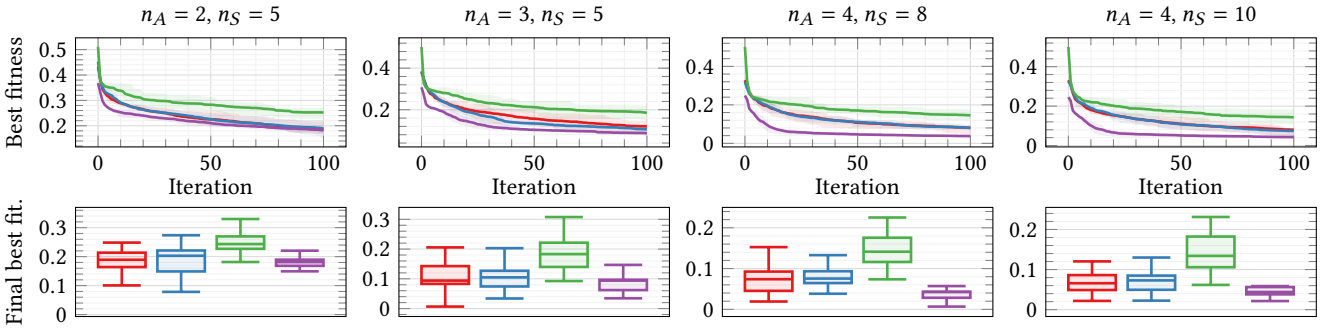

Figure 2: Results for the text extraction domain. Our approach with adaptation and Q is ▭, without adaptation and Q is ▭, without adaptation and Q_{small} is ▭; the baseline is ▭.

of graphs, we look for an extractor in the form of a deterministic finite automaton (DFA) which is, indeed, a graph.

We used four different synthetic problem instances, i.e., pairs s, S, the same ones used in [7]. They differ in the size n_A of the alphabet of symbols composing the string s and the number n_S of substrings in S. In each problem instance, S contains all the substrings that match one or more of the following regular expressions: `000000`, `111(00)?+(11)++`, and `(110110)++`. We built the string s by setting the alphabet A to the first (zero-based) n_A digits and adding random symbols of the alphabet to s until a number of at least n_S substrings of s matched the regular expressions. We considered four datasets obtained using the values $(n_A, n_S) \in \{(2, 5), (3, 5), (4, 8), (4, 10)\}$.

As fitness function, we used the extractor error rate measured on symbols. That is, we counted as an error every character that was not extracted, but belonged to a string of S, and every extracted character that did not belong to a string of S; finally, we computed the error rate as the number of errors divided by the length of s.

5.2.1 Baseline. We used the technique proposed in [7] as baseline. As discussed in Section 1, GraphEA is conceptually similar to our approach, since it operates on graphs and is somehow customizable. However, it requires a direct manipulation of the (code of the) EA to add new domain-specific genetic operators, differently from our case.

In [7], GraphEA has been used for evolving graphs representing DFA and proved to be more effective than grammar-based GP, also because it employs a speciation scheme to protect structural innovations caused by genetic operators. For this reason, we consider GraphEA a challenging baseline.

We used the EA parameter values of [7], with the exception of the n_{pop} and n_{gen}, which we set as for our approach.

5.2.2 Customization of our approach. We set C_G to describe DFA suitable for the alphabet of the problem instance at hand: exactly one node has an attribute marking it as starting node; all nodes can have an attribute marking them as accepting nodes. Edges have a unique attribute assuming as values non empty lists of alphabet symbols. We enforced in C_G all the other structural constraints meaningful for DFA.

Concerning C_{g_0}, we set it to describe a DFA having a unique node, being both the starting node and an accepting node, and a unique looping edge marked with all the possible input symbols.

Finally, concerning Q_{small} and Q, we set them as follows. In Q_{small} we put: (1) one query that removes a node; (2) one that moves a symbol from one edge to another edge starting from the same node; (3) one that flips the accepting attribute of a node; (4) one that changes the starting node; (5) one that adds an edge, with random symbols; (6) one that adds a node without any edges. In Q we inserted all the items of Q_{small} and: (7) one query that adds a node with an incoming edge; (8) one that adds a node with an incoming edge and an outgoing edge; (9) one that adds one edge to each node without an outgoing edge, with random symbols; (10) one that removes an edge; (11) one that changes the target node of an edge;

For the size range of the initial graphs, we used $[2, 82]$.

5.2.3 Results. Figure 2 shows the results of the experiments in text extraction; the figure is organized in the same way of Figure 1.

As in the previous case, the main observation here is about the comparison of our approach with the baseline. In this domain, our approach is not worse than the baseline in two on four problem instance; it is worse on the instances with $n_A = 4$, for which $p < 0.0001$.

Besides the final fitness, our approach looks also slightly slower in convergence; on the other hand, both the adaptive version and the non-adaptive with the full Q are significantly better (and faster in convergence) than the non-adaptive version with Q_{small}. We interpret this finding as a further confirmation that the possibility of easily adding new operators is beneficial: in principle, with other custom operators, e.g., one for each symbol of the alphabet, our approach might match the baseline performance.

5.3 Impact of the adaptation parameters

We conducted two further suites of experiments for assessing the impact of the two main parameters of the adaptation mechanism of our approach: the operator ranking proportion ρ and the schedule for the adaptation rate η.

Concerning the former, we considered the value used in the main variant, $\rho = \frac{1}{3}$, and two other values: $\frac{1}{4}$ and $\frac{1}{2}$. The case $\rho = \frac{1}{2}$ corresponds to updating all the operator weights after each generation, the first half of the rank by increasing them, the other half by decreasing them. The case $\rho = \frac{1}{4}$ leaves instead half of the

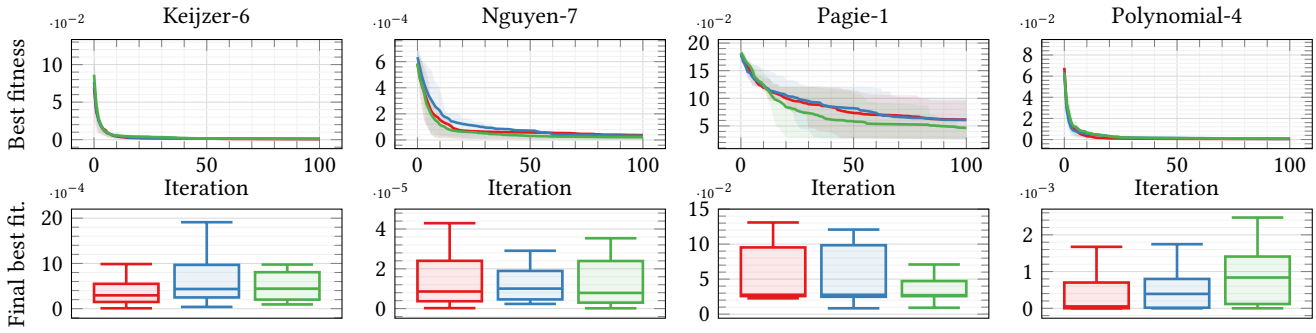

Figure 3: Results for different values of ρ. Our approach, i.e., $\rho = \frac{1}{3}$ is , $\rho = \frac{1}{4}$ is , $\rho = \frac{1}{2}$ is .

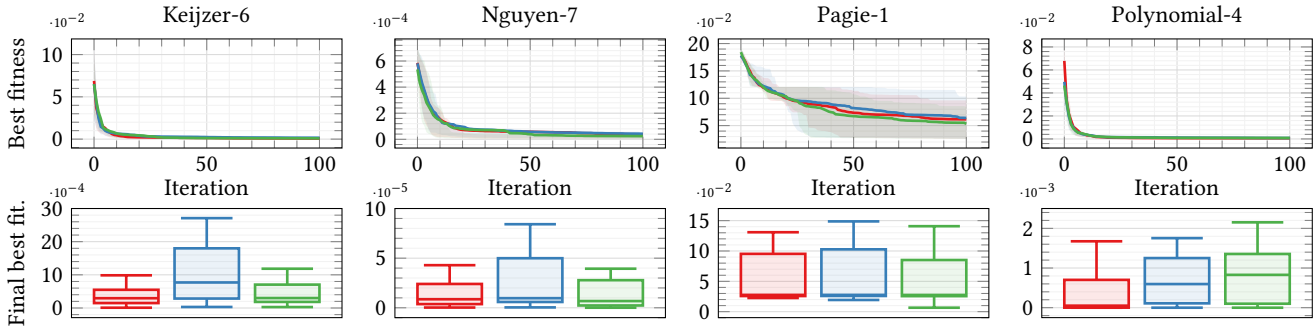

Figure 4: Results for different schedules of η. Our approach, i.e., $\eta = 0.01$ for the entire evolution is , $\eta = 0$ for the first half of the evolution and then $\eta = 0.01$ is , $\eta = 0.01$ for the first half of the evolution and then $\eta = 0$ is .

weights to their default value. The three values for ρ correspond hence to how greedy the adaptation is.

Concerning the schedule for η, i.e., how it varies during the evolution, we considered two alternatives to the schedule used in the main variant, where η remain constant at 0.01. Namely, we considered a case in which $\eta = 0.01$ for the first half of the evolution and then goes to 0, and the opposite one, in which it stays at 0 for the first half of the evolution and then goes to 0.01. In principle, the latter two schedules should affect differently the exploration-exploitation trade-off, since they change the probabilities of the operators (including those that are apparently not effective) in different stages of the evolution.

For this supplementary analysis, we considered four of the five problem instances of the symbolic regression domain. In all cases, we used the full Q.

Figures 3 and 4 show the results, respectively for ρ and η schedule, with the same visual organization of Figures 1 and 2.

The main finding of these experiments is that our approach appears to be robust with respect to the considered parameters, ρ and the schedule for η. Indeed, the differences in the final best fitness are almost never significant, with a few exceptions. For ρ, our default value is better ($p < 0.01$) than $\rho = \frac{1}{2}$ on the Polynomial-4 problem. For η schedule, the constant is better ($p < 0.01$) than the other ones in Keijzer-6 and better ($p < 0.01$) than the one with the adaptation just in the second half of the evolution in Polynomial-4.

While these results suggest that our simple adaptation strategy is effective, we speculate that more sophisticated mechanisms might be better and leave the investigation for future work.

6 CONCLUDING REMARKS

We proposed a Prolog-based representation template for graphs, including genetic operators, and an adaptive EA that together permit the evolutionary optimization on different domains. In practice, the user specifies the domain by means of Prolog "code", without the need of intervening on the EA.

Experimentally, we showed that our approach is versatile and, in general, not worse than other domain-tailored approaches. We also found that evolving graphs with a large number of genetic operators, a possibility enabled by our customizable representation, is effective and that our adaptive EA can cope with the abundance of operators without losing in effectiveness. Moreover, the simple adaptive approach we proposed seems to be robust with respect to the choice of its parameters. Nevertheless, we plan to focus on more sophisticated adaptation strategies in the future.

REFERENCES

[1] Timothy Atkinson. 2019. *Evolving graphs by graph programming.* Ph. D. Dissertation. University of York.

[2] Timothy Atkinson, Detlef Plump, and Susan Stepney. 2018. Evolving graphs by graph programming. In *Genetic Programming: 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings.* Springer, 35–51.

[3] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Giovanni Squillero. 2019. Multi-level diversity promotion strategies for grammar-guided genetic programming. *Applied Soft Computing* 83 (2019), 105599.

[4] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. 2017. Active learning of regular expressions for entity extraction. *IEEE transactions on cybernetics* 48, 3 (2017), 1067–1080.

[5] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabrício Olivetti de França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason H Moore. 2021. Contemporary symbolic regression methods and their relative performance. *arXiv preprint arXiv:2107.14351* (2021).

[6] Dazhuang Liu, Marco Virgolin, Tanja Alderliesten, and Peter AN Bosman. 2022. Evolvability degeneration in multi-objective genetic programming for symbolic regression. In *Proceedings of the Genetic and Evolutionary Computation Conference*. 973–981.

[7] Eric Medvet and Alberto Bartoli. 2021. Evolutionary optimization of graphs with GraphEA. In *AIxIA 2020–Advances in Artificial Intelligence: XIXth International Conference of the Italian Association for Artificial Intelligence, Virtual Event, November 25–27, 2020, Revised Selected Papers*. Springer, 83–98.

[8] Eric Medvet, Fabio Daolio, and Danny Tagliapietra. 2017. Evolvability in grammatical evolution. In *Proceedings of the genetic and evolutionary computation conference*. 977–984.

[9] Eric Medvet, Giorgia Nadizar, and Luca Manzoni. 2022. JGEA: a Modular Java Framework for Experimenting with Evolutionary Computation. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*. 2009–2018.

[10] Julian Francis Miller and Simon L Harding. 2008. Cartesian genetic programming. In *Proceedings of the 10th annual conference companion on Genetic and evolutionary computation*. 2701–2726.

[11] Evgenia Papavasileiou, Jan Cornelis, and Bart Jansen. 2021. A systematic literature review of the successors of "neuroevolution of augmenting topologies". *Evolutionary Computation* 29, 1 (2021), 1–73.

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

[13] Kenneth O Stanley and Risto Miikkulainen. 2002. Evolving neural networks through augmenting topologies. *Evolutionary computation* 10, 2 (2002), 99–127.

[14] Marco Virgolin, Tanja Alderliesten, and Peter AN Bosman. 2019. Linear scaling with and within semantic backpropagation-based genetic programming for symbolic regression. In *Proceedings of the genetic and evolutionary computation conference*. 1084–1092.

[15] David R White, James McDermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. 2013. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines* 14 (2013), 3–29.

[16] Jan Wielemaker, Tom Schrijvers, Markus Triska, and Torbjörn Lager. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2 (2012), 67–96.