Automatic Algebraic Evolutionary Algorithms

Marco Baioletti[1], Alfredo Milani[1,2], and Valentino Santucci[1(✉)]

[1] Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy
{marco.baioletti,alfredo.milani,valentino.santucci}@unipg.it
[2] Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, Hong Kong

Abstract. Motivated from the previously proposed algebraic framework for combinatorial optimization, here we introduce a novel formal languages-based perspective on discrete search spaces that allows to automatically derive algebraic evolutionary algorithms. The practical effect of the proposed approach is that the algorithm designer does not need to choose a solutions encoding and implement algorithmic procedures. Indeed, he/she only has to provide the group presentation of the discrete solutions of the problem at hand. Then, the proposed mechanism allows to automatically derive concrete implementations of a chosen evolutionary algorithms. Theoretical guarantees about the feasibility of the proposed approach are provided.

Keywords: Algebraic evolutionary algorithms
Combinatorial optimization · Formal language perspective

1 Introduction

In a previous series of articles [1,3,11,13], we have introduced an abstract algebraic framework for combinatorial optimization problems. The framework allows to encode in algebraic terms the geometry of the search moves performed by a large class of evolutionary algorithms on the search space of combinatorial problems.

Concrete implementations of the framework have been proposed for discrete spaces such as the permutations and bit-string spaces. Hence, algebraic evolutionary algorithms, such as algebraic differential evolution and particle swarm optimization, have been proposed [1,11]. Interestingly, state-of-the-art and very competitive results have been obtained for permutation flowshop scheduling problems [11,12] and linear ordering problems [2,3].

The main achievement of the algebraic framework is the proposal of abstract definitions for operators that allow to combine and operate on the discrete solutions of the problem at hand. In particular, the proposed operations are addition, subtraction and scalar multiplication. Some abstract algebraic and

M. Pelillo et al. (Eds.): WIVACE 2017, CCIS 830, pp. 271–283, 2018.
https://doi.org/10.1007/978-3-319-78658-2_20

geometric properties, derived from group theory, guarantee that their effects on the involved discrete solutions are geometrically similar to what happen in the classical Euclidean space.

However, the definitions are merely abstract and the algorithm designer needs to instantiate them for any finitely generated group at hand. For instance, randomized decomposer have to be (and have been) provided for the groups of permutations and bit-strings. As an additional, though secondary, result, here we also show how the search space of integer vectors can be represented in the framework.

In this paper we further evolve the framework by proposing general implementations of the abstract operators that are no more abstract but directly operative on any search space that respect some conditions, i.e., to be representable by a finitely presented group. To achieve this aim, we consider a formal language-perspective directly derived from advanced group theory concepts. Therefore, we provide a mechanism to automatically derive operative and universal implementations of the previously proposed algebraic operators by exploiting the concept of group presentation. Discrete solutions are represented as strings of an alphabet (of generators). By changing the alphabet and the equivalence relations on the strings, it is possible to use the Knuth-Bendix completion algorithm [9] to automatically derive concrete operators on different types of solutions (permutations, bit-strings, etc.).

Practically, we make easy the work of the algorithm designer that can now avoid to choose a solutions encoding and implement the abstract procedures of the framework for this encoding. Note anyway, that this proposal is a sort of "proof of concept". Indeed, we do not provide any experimental result, but only theoretical guarantees about the feasibility of the proposed implementations.

The rest of the paper is organized as follows. Section 2 describes the previously proposed abstract algebraic framework together with some of its concrete implementations. The algebraic evolutionary operators are then derived in Sect. 3. The core of the paper is represented by Sects. 4 and 5 where we provide, respectively, theoretical foundations of the language-based perspective, and the concrete and general algorithmic implementations. Finally, Sect. 6 concludes the paper by also providing some future lines of research.

2 Abstract Algebraic Framework

In this section we provide a concise description of the algebraic framework for evolutionary computation previously proposed in [11], together with its extension introduced in [3]. The framework is based on the notion of *finitely generated group* and the related algebraic and geometric concepts. Its aim is to introduce the operations $\oplus$, $\ominus$, $\odot$ on the set of discrete solutions in such a way that they simulate, as much as possible, the analogous vector operations of the Euclidean space.

2.1 Search Spaces and Finitely Generated Groups

The triplet $G = (X, \star, H)$ is a finitely generated group representing a combinatorial search space if and only if:

- X is the discrete set of solutions in the search space;
- $\star : X \times X \to X$ is a binary operation on X which satisfies the group properties: associativity, existence of the identity $e \in X$, and existence of the inverse $x^{-1} \in X$ for any $x \in X$; if $\star$ is also commutative, the group is Abelian, but it is not required;
- $H \subseteq X$ is a finite generating set of the group, i.e., any $x \in X$ can be decomposed as $x = h_1 \star \cdots \star h_l$ for some $h_1, \ldots, h_l \in H$.

A decomposition $x = h_1 \star \cdots \star h_l$ of $x \in X$ is minimal if there exists no other decomposition $x = h'_1 \star \cdots \star h'_m$ with $m < l$. The length l of a minimal decomposition of x is the weight of x and it is denoted by $|x|$.

Given a finitely generated group $G = (X, \star, H)$, its Cayley graph $\mathcal{C}(G)$ is the labelled digraph whose vertexes are the solutions in X and there exists an arc from x to y labelled by $h \in H$ if and only if $y = x \star h$.

In the Cayley graph, for all $x \in X$, every directed path from e to x corresponds to a decomposition of x: if the arcs labels occurring in the path are $\langle h_1, h_2, \ldots, h_l \rangle$, then $x = h_1 \star h_2 \star \cdots \star h_l$. As a consequence, shortest paths from e to x correspond to minimal decompositions of x. More generally, a shortest path from x to y, where $x, y \in X$, corresponds to a minimal sequence of generators $\langle h_1, h_2, \ldots, h_l \rangle$ such that $x \star (h_1 \star h_2 \star \cdots \star h_l) = y$. Hence, $\langle h_1, h_2, \ldots, h_l \rangle$ is a minimal decomposition of $x^{-1} \star y$.

The diameter D of $\mathcal{C}(G)$ is defined as the maximal weight of the elements in X. Moreover, an interesting partial order relation, which will be useful later, is defined as follows. For $x, y \in X$, $x \sqsubseteq y$ if and only if there exists (at least) a shortest path from e to y passing by x. For the sake of presentation, here we focus on groups with a unique maximal weight element ω such that $x \sqsubseteq \omega$ for all $x \in X$. The concrete group considered later belongs to such a class.

The Cayley graph has an important geometric interpretation. Indeed, a sequence of generators $\langle h_1, h_2, \ldots, h_l \rangle$ can be seen as a *vector* which connects a starting *point* $x \in X$ to the end *point* $y = x \star (h_1 \star h_2 \star \cdots \star h_l)$. On the other hand, any element $x \in X$ can be decomposed as a sequence of generators $\langle h_1, h_2, \ldots, h_l \rangle$ and therefore it can be considered also as a *free vector*. The dichotomous interpretation of the elements of X, as points and as vectors, allows to define the operations $\oplus, \ominus, \odot$ on X which simulate the analogous operations of the Euclidean space.

2.2 Addition and Subtraction

The addition $z = x \oplus y$ is defined as the application of the vector $y \in X$ to the point $x \in X$. The result z is computed by choosing a decomposition $\langle h_1, h_2, \ldots, h_l \rangle$ of y and by finding the end point of the path which starts from x and whose arcs labels are $\langle h_1, h_2, \ldots, h_l \rangle$, i.e., $z = x \star (h_1 \star h_2 \star \cdots \star h_l)$. By noting

that $h_1 \star h_2 \star \cdots \star h_l = y$, the addition $\oplus$ is independent from the generating set and is uniquely defined as

$$x \oplus y := x \star y. \tag{1}$$

Continuing the analogy with the Euclidean space, the difference between two points is a vector. Given $x, y \in X$, the difference $y \ominus x$ produces the sequence of labels $\langle h_1, h_2, \ldots, h_l \rangle$ in a path from x to y. Since $h_1 \star h_2 \star \cdots \star h_l = x^{-1} \star y$, we can replace the sequence of labels with its product, thus making the difference independent from the generating set. Therefore, $\ominus$ is uniquely defined as

$$y \ominus x := x^{-1} \star y. \tag{2}$$

Both $\oplus$ and $\ominus$, like their numerical counterparts, are consistent to each other. Indeed, $x \oplus (y \ominus x) = y$ for all $x, y \in X$. Moreover, both operations are not commutative (unless the group is Abelian), $\oplus$ is associative, and e is its neutral element.

2.3 Scalar Multiplication

Again, as in the Euclidean space, it is possible to multiply a vector by a non-negative scalar. Given $a \geq 0$ and $x \in X$, we denote their multiplication with $a \odot x$.

We first provide the conditions that $a \odot x$ has to verify in order to simulate, as much as possible, the scalar multiplication of vector spaces:

(C1) $|a \odot x| = \lceil a \cdot |x| \rceil$;
(C2) if $a \in [0, 1]$, $a \odot x \sqsubseteq x$;
(C3) if $a \geq 1$, $x \sqsubseteq a \odot x$.

Clearly, the scalar multiplication of $\mathbb{R}^n$ satisfies the slight variant of (C1) where the Euclidean norm replaces the group weight and the ceiling is omitted. Besides, similarly to scaled vectors in $\mathbb{R}^n$, (C2) and (C3) intuitively encode the idea that $a \odot x$ is the element x scaled down or up, respectively.

It is important to note that, fixed a and x, there may be more than one element of X satisfying (C1–C3). This is a clear consequence of the non uniqueness of the minimal decomposition of x. Therefore, different strategies can be devised to compute $a \odot x$. Nevertheless, our aim is to apply the operation in evolutionary algorithms, therefore we denote with $a \odot x$ a randomly selected element satisfying (C1–C3).

Note also that the diameter D induces an upper bound on the possible values for the scalar a. Indeed, for any $x \in X$, let $\overline{a}_x = \frac{D}{|x|}$, if $a > \overline{a}_x$, (C1) would imply $|a \odot x| > D$, but this is impossible. Therefore, similarly to out-of-bounds handling techniques of continuous evolutionary algorithms, we define

$$a \odot x := \overline{a}_x \odot x, \text{ when } a > \overline{a}_x. \tag{3}$$

The multiplication $a \odot x$ can be computed by: (i) randomly selecting a shortest path from e to ω passing by x, and (ii) composing the first $\lceil a \cdot |x| \rceil$ generators

on its arcs. Since any sub-path of a shortest path is itself a shortest path, and by also considering that shortest paths correspond to minimal decompositions, it is easy to see that the conditions (C1–C3) are satisfied.

Let $l = |x|$, we can observe that the sequence of generators $\langle h_1, \ldots, h_l, \ldots, h_D \rangle$ on the chosen shortest path can be divided in two parts: $\langle h_1, \ldots, h_l \rangle$ and $\langle h_{l+1}, \ldots, h_D \rangle$. The former is a minimal decomposition of x, while the latter minimally decomposes $x^{-1} \star \omega$. Operatively, only one of the sub-paths is used to compute $a \odot x$. When $a \leq 1$, the generators to compose are all in the first sub-path $\langle h_1, \ldots, h_l \rangle$. Conversely, for $a > 1$, it is sufficient to take the first $\lceil a \cdot l \rceil - l$ generators in the second sub-path $\langle h_{l+1}, \ldots, h_D \rangle$ and compose them to the right of x.

The pseudo-codes of the two procedures for $a \in [0, 1]$ and $a > 1$ are reported, respectively, in Figs. 1 and 2. Both rely on the abstract procedure *RandDec* which is assumed to return a random minimal decomposition of the element in input. An implementation of *RandDec* has to consider the particularities of the concrete finitely generated group at hand. Note also that *Extend* implements Eq. (3).

```
1: function TRUNCATE(a ∈ [0, 1], x ∈ X)
2:     s ← RandDec(x)
3:     l ← Length(s)
4:     k ← ⌈a · l⌉
5:     z ← e
6:     for i ← 1 to k do
7:         z ← z ⋆ s_i
8:     end for
9:     return z
10: end function
```

Fig. 1. Truncation algorithm for computing $a \odot x$ when $a \in [0, 1]$

```
1: function EXTEND(a > 1, x ∈ X)
2:     s ← RandDec(x^{-1} ⋆ ω)
3:     l ← D − Length(s)
4:     ā_x = D/l
5:     a ← min{a, ā_x}
6:     k ← ⌈a · l⌉
7:     z ← x
8:     for i ← 1 to k − l do
9:         z ← z ⋆ s_i
10:    end for
11:    return z
12: end function
```

Fig. 2. Extension algorithm for computing $a \odot \pi$ when $a > 1$

2.4 Concrete Implementations

Given a concrete finitely generated group (FGG) modeling the search space at hand, in order to implement the abstract vector operations described in Sects. 2.2 and 2.3, it is sufficient to provide procedures to: (i) invert an element (x^{-1}), (ii) compose two elements ($x \star y$), (iii) randomly decompose an element in terms of the generators (*RandDec*). Moreover, note that the procedures for (i) and (ii) are usually straightforward.

Three concrete FGGs that allow to cover the vast majority of the combinatorial optimization problems are: the group of the n-length bit-strings $\mathbb{B}^n$, the group of the n-length permutations $\mathcal{S}_n$, and the group of the n-length integer vectors $\mathbb{Z}^n$.

The bit-strings in $\mathbb{B}^n$ form a group by considering the classical bitwise XOR operator $\veebar$. The generators are the strings with one 1-bit and $n-1$ 0-bits. Therefore, computing a random decomposition of a given bit-string simply reduces to choosing an ordering (i.e., a permutation) of its 1-bits. Note also that, given a generic $x \in \mathbb{B}^n$ and the generator u_i (i.e., the bit-string with only one 1-bit at position i), the composition $x \veebar u_i$ practically corresponds to flip the i-bit of x. Hence, the induced Cayley graph and distance correspond to classical concepts such as, respectively, the binary hypercube and the Hamming distance.

All the permutations of the set $[n] = \{1, \ldots, n\}$ form the "symmetric group" $\mathcal{S}_n$ by considering the classical permutation composition operator $\circ$ defined as $(\pi \circ \rho)(i) = \pi(\rho(i))$ for all items $i \in [n]$ and $\pi, \rho \in \mathcal{S}_n$. Different generating sets are possible in $\mathcal{S}_n$ (see [3,14]). The simplest is the subset of the $n-1$ simple transpositions, i.e., the set $ST = \{\sigma_i \in \mathcal{S}_n : 1 \leq i < n\}$ where σ_i is defined as: $\sigma_i(i) = i+1$, $\sigma_i(i+1) = i$, and $\sigma_i(j) = j$ for $j \in [n] \setminus \{i, i+1\}$. Given a generic $\pi \in \mathcal{S}_n$, the composition $\pi \circ \sigma_i$ corresponds to swap the adjacent items i and $i+1$ in π. Hence, by modifying the classical bubble sort algorithm, in [11] we have provided a randomized decomposer for $\mathcal{S}_n$. Moreover, other interesting generators are those which encode exchange and insertion moves of generic items in the permutation. Implementations of these generating sets have been discussed and provided in [3].

Finally, the integer vectors in $\mathbb{Z}^n$ form a group by considering the classical arithmetic addition $+$. In this case, the generators are the n-length vectors formed by $n-1$ zeros and one component equal to ± 1. A randomized decomposer for $\mathbb{Z}^n$ is straightforward to derive. Note also that this group, differently from the other ones, is infinite and it does not have a maximum weight element. Apparently, this does not allow to implement the algorithm *Extend* of Fig. 2. Anyway, a simple generalization of *Extend* fixes the problem. The idea is to iteratively choose a random generators among all the generators that increase the group weight of the current element. In $\mathbb{Z}^n$, the group weight is the arithmetic sum of the vector components.

3 Algebraic Evolutionary Operators

It is possible to straightforwardly derive algebraic evolutionary operators by using the operations introduced in Sect. 2 in order to redefine the move equations of the most popular evolutionary and swarm intelligence algorithms for continuous optimization.

Here we provide the formal redefinitions for: the mutation operator of Differential Evolution (DE) [15], the velocity and position update equations of Particle Swarm Optimization (PSO) [8], and the update equation of the Firefly Algorithm (FA) [18]. The first two have been proposed in, respectively, [1,11], while the third is a novelty of this work. The following definitions subsume that a finitely generated group X is given.

The differential mutation of DE, given three distinct population individuals $x_0, x_1, x_2 \in X$ and a scalar $F \in [0, 1]$, generates a mutant $u \in X$ according to

$$u \leftarrow x_0 \oplus F \odot (x_1 \ominus x_2). \tag{4}$$

A PSO particle is formed by its current position x, velocity v, personal and social best positions p and g. All these particle's properties can be encoded by using group elements, i.e., $x, v, p, g \in X$. Hence, given also the three scalar parameters $w, c_1, c_2 \geq 0$, the particle's new velocity $v' \in X$ and position $x' \in X$ are computed according to

$$v' \leftarrow [w \odot v] \oplus [(r_1 c_1) \odot (p \ominus x)] \oplus [(r_2 c_2) \odot (g \ominus x)], \tag{5}$$

$$x' \leftarrow x \oplus v', \tag{6}$$

where r_1, r_2 are two randomly generated numbers in $[0, 1]$.

In FA, the i-th computational firefly updates its position $x_i \in X$ by moving towards the positions of the brighter fireflies $j_1, \ldots, j_k$, and by considering fitness as brightness. Formally, the new position x'_i is computed according to

$$x'_i \leftarrow x_i \oplus \bigoplus_{h=1}^{k} \left[\beta_0 \exp\left(-\gamma d(x_i, x_{j_h})^2\right) \odot (x_{j_h} \ominus x_i) \oplus (\alpha \odot \epsilon)\right], \tag{7}$$

where $\alpha, \beta_0, \gamma \geq 0$ are the FA scalar parameters, d is the distance induced by the finitely generated group at hand, and ϵ is a randomly generated discrete solution. Note how, with respect to the previous case, the FA update rule makes an explicit use of the discrete distance function induced by the finitely generated group at hand.

Generally, when the group in not Abelian, the composition is not independent of the terms ordering. This issue has been addressed in [3]. Finally note that many other evolutionary algorithms for numerical optimization can be adapted for combinatorial search spaces using our framework. Some examples are: artificial bee colony [7], bacterial foraging optimization [5], cuckoo search [17], and the fireworks algorithm [16].

4 A Formal Language Perspective

A formal language perspective on the algebraic framework described in Sect. 2 can be introduced by restricting our focus to a sub-class of finitely generated groups, namely, the finitely presented groups. All the concrete groups discussed in Sect. 2.4 are finitely presented. Moreover, other popular groups, like for example the braid group [6], are usually directly threated by means of their presentation. Hence, restricting to finitely presented groups does not result in any practical issue for our purposes.

Formally, the group $(X, \star)$ is finitely presented if there exists a presentation (H, R) such that: (i) $H \subseteq X$ generates X, i.e., $(X, \star)$ is finitely generated by the generating set H; (ii) R is a finite set of equivalence relations (made using the group operation $\star$) among the generators in H.

Interestingly, the presentation (H, R) of the group $(X, \star)$ allows to interpret:

1. the generators in H as a set of symbols, i.e., an alphabet,
2. the elements in X as strings over the alphabet H, i.e., $x \in X$ if and only if $x \in H^*$,
3. the group operation $\star$ as a concatenation of strings, i.e., given $x, y \in H^*$ then $x \star y = xy$, where xy denotes the concatenation of x and y; and
4. the equivalence relations in R as rewriting rules for equivalent strings, i.e., if $(v, w) \in R$ then $vxw = vxv = wxv = wxw \in H^*$.

This formal language perspective allows to introduce a further level of generalization in our framework. Practically, we can facilitate the work of the algorithm designer by avoiding him/her to peak up a solutions' representation for the problem at hand. Indeed, the goal of this section is to show how a generic group presentation can be used to automatically derive generic implementations of three operators $\oplus, \ominus, \odot$.

The main idea is to encode the group elements (i.e., the solutions of the problem at hand) by means of their representation as strings of generators. Then, $\oplus, \ominus, \odot$ can be automatically derived by simply introducing general procedures for element's inversion, composition and random decomposition that work directly on the string representation of the element.

By considering a generating set closed for inversions, i.e., $h \in H$ if and only if $h^{-1} \in H$, the inversion can be straightforwardly derived by exploiting the basic group properties. Formally, given a generic string $h_{i_1} h_{i_2} \dots h_{i_l} \in H^*$, its inverse is defined as

$$INV(h_{i_1} h_{i_2} \dots h_{i_l}) := h_{i_l}^{-1} \dots h_{i_2}^{-1} h_{i_l}^{-1}. \tag{8}$$

Composition, as already explained in the point 3 above, becomes a simple concatenation of strings. Formally, given $x, y \in H^*$:

$$CONCAT(x, y) := xy. \tag{9}$$

More interesting is the operation of random minimal decomposition *RandDec*. First note that a string representation of a group element is already

a decomposition in terms of generators, thus the problem becomes to find a procedure to simplify the string as much as possible. Intuitively, we can iteratively apply the equivalences in R in order to rewrite a string until it becomes of minimal length. The problem with this approach is that it is not easy to know when to stop. Luckily, the Knuth-Bendix (KB) completion algorithm [9] is a popular tool in computational algebra that allows to solve this issue.

KB takes in input the group presentation (H, R) and produces a terminating and confluent rewriting system RWS for the strings in H^*. RWS is nothing else than a set of rewriting rules such as $v \to w$ that can be iteratively applied to a string s until it does not match any rule in RWS, i.e., s has been reduced to its minimal length. Formally, we denote with $RWS(s)$ the minimal length string obtained by simplifying s with the rules in RWS. KB guarantees that the application of RWS terminates (i.e., it is a terminating system) and that the minimal form obtained for any given input string does not depend on the ordering by which the matching rules have been applied at every rewriting iteration (i.e., RWS is a confluent system). Therefore, KB can be run offline only once, because of that we can store the produced rewriting system RWS (it can be actually provided in different ways, one of them is as a finite state automaton) and execute it every time we need a minimal decomposition.

There is only one last issue. We need a randomized minimal decomposer, but the rewriting system produced by KB is confluent, i.e., deterministic. In order to introduce randomization, let consider that KB needs, as additional input, an arbitrary ordering (i.e., a permutation) of the generators in H. Hence, feeding KB with two different orderings on H produces two distinct rewriting systems RWS_1 and RWS_2 that both are terminating and confluent, but such that, in general, $RWS_1(s) \neq RWS_2(s)$. Therefore, in the offline computation stage, we can peak up k different permutations of H and run KB for k times in order to obtain k different rewriting system $RWS_1, \ldots, RWS_k$. Then, a random minimal decomposition of a string $x \in H^*$ can be computed as

$$RandDec(x) := RWS_r(x), \tag{10}$$

where r is a random integer in $[1, k]$.

Summarizing, the (finite) presentation of a group allows to: (i) represent the group elements as string of generators, thus that no group (or problem) dependent encoding has to be considered, and (ii) provide general concrete implementations (i.e., working on any possible finitely presented group) of the element's inversion, composition and random minimal decomposition.

5 Automatic Algebraic Evolutionary Algorithms

Here we show how it is possible to automatically generate the implementation of an algebraic evolutionary algorithms by starting from a given group presentation.

By using the language-based tools provided in Sect. 4 we provide generic, but operative, implementations of the operators $\oplus, \ominus, \odot$ for any possible group presentation.

Let $x, y \in H^*$ be two group elements represented as strings of generators. Then, addition and subtraction are defined according to, respectively,

$$x \oplus y := CONCAT(x, y), \tag{11}$$

and

$$x \ominus y := CONCAT(INV(y), x). \tag{12}$$

For the scalar multiplication $\odot$ we provide language-based implementations of *Truncate* and *Extend* in, respectively, Figs. 3 and 4.

```
1: function TRUNCATE(a ∈ [0, 1], x ∈ H*)
2:     x' ← RandDec(x)                    ▷ RandDec is defined in equation (10)
3:     l ← Length(x')
4:     k ← ⌈a · l⌉
5:     z ← ϵ                              ▷ ϵ ∈ H* is the empty string
6:     for i ← 1 to k do
7:         z ← CONCAT(z, x'_i)            ▷ x'_i is the i-th generators of x'
8:     end for
9:     return z
10: end function
```

Fig. 3. Generic implementation of the truncation algorithm

```
1: function EXTEND(a > 1, x ∈ H*)
2:     RWS ← a randomly choosen rewriting system from {RWS_1, ..., RWS_k}
3:     x' ← RWS(x)
4:     l ← Length(x')
5:     k ← ⌈a · l⌉
6:     z ← x'
7:     for i ← 1 to k − l do
8:         h ← a randomly generator such that len(RWS(zh)) = len(RWS(z)) + 1
9:         z ← CONCAT(z, h)
10:    end for
11:    return z
12: end function
```

Fig. 4. Generic implementation of the extension algorithm

Moreover, note that some algebraic evolutionary operators also needs the computation of the group distance $d(x, y)$ (see for example Eq. (7)). However, it is easy to show that $d(x, y) = \text{Length}(RandDec(x \ominus y))$.

Therefore, it is now straightforward to show how, using the language-based implementations of $\oplus, \ominus, \odot$, we can automatically derive an algorithm implementation by simply providing a group presentation and choosing the preferred

algebraic algorithmic schemes (e.g., the algebraic PSO described by Eqs. (5) and (6)). Figure 5 depicts and summarizes the main idea of this approach. Given a combinatorial problem to solve, the algorithm designer does not need to choose a solutions encoding. Indeed, he/she only needs to: provide a fitness function, choose its preferred algorithmic schemes (that uses algebraic operators), and provide a group presentation for the problem at hand. Note that the last step is usually straightforward, since group presentations for the set of problem solution is often directly available. The group presentation is then used in an offline computational stage where KB algorithm generates the rewriting systems $RWS_1, \ldots, RWS_k$. Then, both the group presentation and the generated rewriting systems are used by the general implementations of $\oplus, \ominus, \odot$ that, in turn, allow to obtain the desired evolutionary behavior. In conclusion, the proposed automatic mechanism substantially reduces the work of the algorithm's designer.

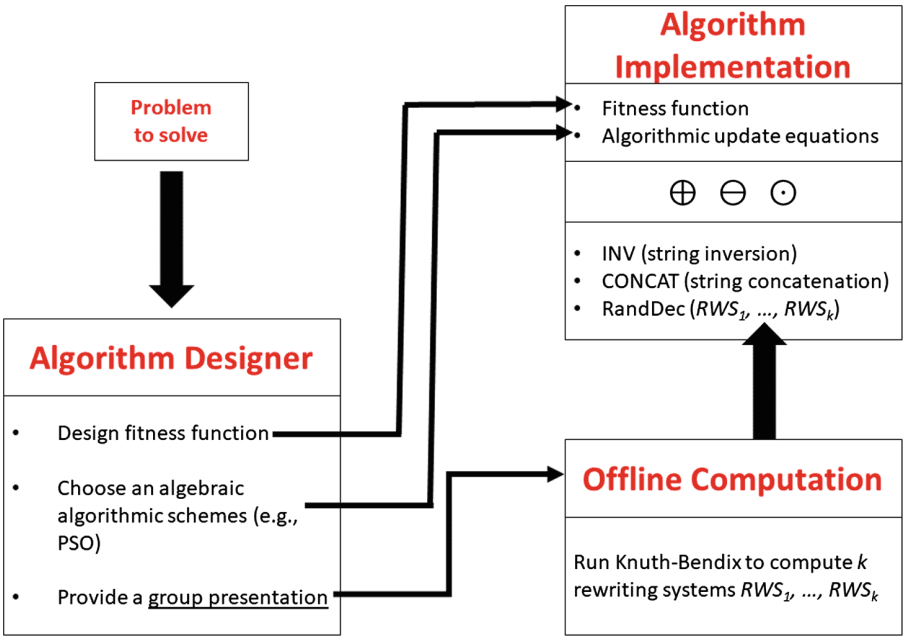

Fig. 5. Automatic generation of an algebraic evolutionary algorithm

6 Conclusion and Future Work

Starting from the algebraic framework for combinatorial optimization previously proposed in [1,4,11,12], in this paper we have provided a mechanism to automatically derive concrete implementations of the framework for any search space representable by a finitely presented group.

To achieve this goal, a formal language perspective on the search space has been introduced. The main algebraic tool employed is the well known Knuth-Bendix completion algorithm.

As a future line of research we will consider the implementation of our proposal to derive algebraic evolutionary algorithms in order to address braid optimization problems [6] that have applications in the field of quantum computing, see for example [10].

References

1. Baioletti, M., Milani, A., Santucci, V.: Algebraic particle swarm optimization for the permutations search space. In: Proceedings of IEEE Congress on Evolutionary Computation, CEC 2017, pp. 1587–1594 (2017). https://doi.org/10.1109/CEC.2017.7969492
2. Baioletti, M., Milani, A., Santucci, V.: Linear ordering optimization with a combinatorial differential evolution. In: Proceedings of 2015 IEEE International Conference on Systems, Man, and Cybernetics, SMC 2015, pp. 2135–2140 (2015). https://doi.org/10.1109/SMC.2015.373
3. Baioletti, M., Milani, A., Santucci, V.: An extension of algebraic differential evolution for the linear ordering problem with cumulative costs. In: Handl, J., Hart, E., Lewis, P., López-Ibáñez, M., Ochoa, G., Paechter, B. (eds.) PPSN 2016. LNCS, vol. 9921, pp. 123–133. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-45823-6_12
4. Baioletti, M., Milani, A., Santucci, V.: A new precedence-based ant colony optimization for permutation problems. In: Shi, Y., et al. (eds.) SEAL 2017. LNCS, vol. 10593, pp. 960–971. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68759-9_79
5. Das, S., Biswas, A., Dasgupta, S., Abraham, A.: Bacterial foraging optimization algorithm: theoretical foundations, analysis, and applications. In: Abraham, A., Hassanien, A.E., Siarry, P., Engelbrecht, A. (eds.) Foundations of Computational Intelligence Volume 3. SCI, vol. 203, pp. 23–55. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01085-9_2
6. Garside, F.A.: The braid group and other groups. Q. J. Math. **20**(1), 235–254 (1969)
7. Karaboga, D., Basturk, B.: A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. J. Glob. Optim. **39**(3), 459–471 (2007)
8. Kennedy, J., Eberhart, R.: Particle swarm optimization. In: Proceedings of IEEE International Conference on Neural Networks, vol. 4, pp. 1942–1948 (1995)
9. Knuth, D.E.: The genesis of attribute grammars. In: Deransart, P., Jourdan, M. (eds.) Attribute Grammars and their Applications. LNCS, vol. 461, pp. 1–12. Springer, Heidelberg (1990). https://doi.org/10.1007/3-540-53101-7_1
10. McDonald, R.B., Katzgraber, H.G.: Genetic braid optimization: a heuristic approach to compute quasiparticle braids. Phys. Rev. B **87**(5), 054414 (2013)
11. Santucci, V., Baioletti, M., Milani, A.: Algebraic differential evolution algorithm for the permutation flowshop scheduling problem with total flowtime criterion. IEEE Trans. Evol. Comput. **20**(5), 682–694 (2016). https://doi.org/10.1109/TEVC.2015.2507785

12. Santucci, V., Baioletti, M., Milani, A.: Solving permutation flowshop scheduling problems with a discrete differential evolution algorithm. AI Commun. **29**(2), 269–286 (2016). https://doi.org/10.3233/AIC-150695
13. Santucci, V., Baioletti, M., Milani, A.: A differential evolution algorithm for the permutation flowshop scheduling problem with total flow time criterion. In: Bartz-Beielstein, T., Branke, J., Filipič, B., Smith, J. (eds.) PPSN 2014. LNCS, vol. 8672, pp. 161–170. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-10762-2_16
14. Schiavinotto, T., Stützle, T.: A review of metrics on permutations for search landscape analysis. Comput. Oper. Res. **34**(10), 3143–3153 (2007)
15. Storn, R., Price, K.: Differential evolution - a simple and efficient heuristic for global optimization over continuous spaces. J. Glob. Optim. **11**(4), 341–359 (1997)
16. Tan, Y., Zhu, Y.: Fireworks algorithm for optimization. In: Tan, Y., Shi, Y., Tan, K.C. (eds.) ICSI 2010. LNCS, vol. 6145, pp. 355–364. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13495-1_44
17. Yang, X.S., Deb, S.: Cuckoo search via Levy flights. In: 2009 World Congress on Nature Biologically Inspired Computing (NaBIC), pp. 210–214 (2009)
18. Yang, X.-S.: Firefly algorithms for multimodal optimization. In: Watanabe, O., Zeugmann, T. (eds.) SAGA 2009. LNCS, vol. 5792, pp. 169–178. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04944-6_14