An Algebraic Approach for the Search Space of Permutations with Repetition

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

[1] Department of Mathematics and Computer Science, University of Perugia, Perugia, Italy
{marco.baioletti,alfredo.milani}@unipg.it

[2] Department of Humanities and Social Sciences, University for Foreigners of Perugia, Perugia, Italy
valentino.santucci@unistrapg.it

Abstract. We present an algebraic approach for dealing with combinatorial optimization problems based on permutations with repetition. The approach is an extension of an algebraic framework defined for combinatorial search spaces which can be represented by a group (in the algebraic sense). Since permutations with repetition does not have the group structure, in this work we derive some definitions and we devise discrete operators that allow to design algebraic evolutionary algorithms whose search behavior is in line with the algebraic framework. In particular, a discrete Differential Evolution algorithm which directly works on the space of permutations with repetition is defined and analyzed. As a case of study, an implementation of this algorithm is provided for the Job Shop Scheduling Problem. Experiments have been held on commonly adopted benchmark suites, and they show that the proposed approach obtains competitive results compared to the known optimal objective values.

Keywords: Discrete evolutionary algorithms · Permutations with Repetition · Algebraic approach

1 Introduction

An algebraic framework for combinatorial optimization problems has been previously proposed in a series of articles [4–6, 22]. This framework mainly proposed the discrete operations of sum, difference, and scalar multiplication that allow to design discrete variants of widely used continuous evolutionary algorithms such as the Differential Evolution (DE) [22] and the Particle Swarm Optimization (PSO) [23]. The main requirement of the framework is that the solutions in the search space of the combinatorial problem at hand must form a group (in the algebraic sense). For instance, this is the case of widely considered search spaces such as those of bit-strings [24] and permutations [1, 6].

However, there are interesting problems defined in combinatorial search spaces which do not form a group. One of these is the space of permutations with repetition, i.e., ordering of items which – differently from classical permutations – can appear multiple times in the sequence. This search space has been considered, for example, in [8,14,16,20,27]. The most notable applications of permutations with repetition are in some scheduling and partitioning problems. Indeed, in the scheduling case, the repeated items accommodate the fact that some jobs need to be processed in more than one machine, while, in partitioning problems, permutations with repetition are intended as assignments of items to a particular cluster among a set of clusters with a given size. Widely known example of such problems are the job shop scheduling problem [11] and the balanced multiway graph partitioning problem [17].

In this work, we extend the algebraic framework in order to work on the search space of permutations with repetition, even if they do not form a group. With this regard, we derive formal definitions and algorithmic implementations of the discrete operators of sum, difference and scalar multiplication. Such operators allow to design discrete variants of evolutionary algorithms which are commonly and effectively used in the continuous search spaces. In particular, we introduce a discrete Algebraic Differential Evolution for Permutations with Repetition (ADE-PR) by also analyzing its search behavior.

ADE-PR can in principle be applied to any problems requiring a permutation with repetition as a solution. As a case of study, we have investigated the effectiveness of ADE-PR on the Job Shop Scheduling Problem (JSSP). Therefore, few additional algorithmic components, purposely defined for the JSSP, have been integrated in ADE-PR. Finally, computational experiments have been held by considering widely used benchmark suites for the JSSP.

The rest of the paper is organized as follows. Section 2 recalls the algebraic framework which has been extended in Sect. 3 in order to handle the space of permutations with repetition. Section 4 describes the main scheme of ADE-PR, while its implementation for the JSSP is depicted in Sect. 5. The experimental analysis is described in Sect. 6, while Sect. 7 concludes the paper by also providing future lines of research.

2 Algebraic Background

2.1 The Abstract Algebraic Framework for Evolutionary Computation

The algebraic framework for evolutionary computation, firstly proposed in [22] and further studied in [3,4,6,7], allows to define the discrete operators $\oplus$, $\ominus$ and $\odot$ which simulate in a discrete search space the properties of their numerical counterparts.

In particular, these discrete operators are abstractly defined for any combinatorial search space whose solution set can be represented with an algebraic structure known as *finitely generated group* [18].

The triple $(X, \circ, G)$ is a finitely generated group representing the search space of a given combinatorial optimization problem $\mathcal{P}$ if:

- X is the set of solutions of $\mathcal{P}$;
- $\circ$ is a binary operation on X satisfying the group properties, i.e., closure, associativity, identity (e), and invertibility (x^{-1}); and
- $G \subseteq X$ is a finite generating set of the group, i.e., any $x \in X$ has a (not necessarily unique) minimal-length decomposition $\langle g_1, \ldots, g_l \rangle$, with $g_i \in G$ for all $i \in \{1, \ldots, l\}$, and whose evaluation is x, i.e., $x = g_1 \circ \cdots \circ g_l$.

Moreover, the length of a minimal decomposition of a discrete solution $x \in X$ is denoted by $|x|$.

Using $(X, \circ, G)$, it is possible to provide formal definitions of the operators $\oplus$, $\ominus$ and $\odot$. Let $x, y \in X$ and $\langle g_1, \ldots, g_k, \ldots, g_{|x|} \rangle$ be a minimal decomposition of x, then

$$x \oplus y := x \circ y, \tag{1}$$

$$x \ominus y := y^{-1} \circ x, \tag{2}$$

$$a \odot x := g_1 \circ \cdots \circ g_k, \text{ with } k = \lceil a \cdot |x| \rceil \text{ and } a \in [0, 1]. \tag{3}$$

Interesting graph-based interpretations of these definitions can be given as follows. The algebraic structure on the search space naturally defines neighborhood relations among the solutions. Indeed, any finitely generated group $(X, \circ, G)$ is associated to a labelled digraph $\mathcal{G}$ whose vertices are the solutions in X and two generic solutions $x, y \in X$ are linked by an arc labelled by $g \in G$ if and only if $y = x \circ g$. Therefore, a simple one-step move in the search space can be directly encoded by a generator, while a composite move can be synthesized as the evaluation of a sequence of generators (a path on the graph).

In analogy with $\mathbb{R}^n$, the elements of X can be dichotomously interpreted both as solutions (vertices on the graph) and as displacements between solutions (labelled paths on the graph). As detailed in [22], this allows to provide rational interpretations of the definitions (1), (2) and (3) as follows:

- $x \oplus y$ is the vertex of $\mathcal{G}$ where we arrive if we move from the vertex x following the arcs in any (minimal) decomposition of y;
- a minimal decomposition of $x \ominus y$ corresponds to the sequence of arcs in a shortest path from the vertex y to the vertex x in $\mathcal{G}$;
- the scalar multiplication $a \odot x$, with $a \in [0, 1]$, corresponds to truncating a shortest path from the vertex e (the identity of the group) to the vertex x in $\mathcal{G}$.

Clearly, these geometrical interpretations are in line with the vectors/points interpretations of the classical Euclidean space.

2.2 The Algebraic Differential Evolution

As shown in [22] and [23], expressions which involve the three discrete operators allow to derive discrete variants of some popular evolutionary schemes originally defined for continuous problems [19,26]. For instance, a discrete variant of the Differential Evolution (DE) algorithm, namely Algebraic Differential Evolution

(ADE), can be defined by simply replacing the classical mathematical operations with their discrete variants $\oplus, \ominus, \odot$ in the definition of the differential mutation which is the key operator of the DE.

Therefore, the differential mutation of ADE is defined as follows:

$$v \leftarrow x_{r_0} \oplus F \odot (x_{r_1} \ominus x_{r_2}), \tag{4}$$

where $x_{r_0}, x_{r_1}, x_{r_2} \in X$ are three randomly selected population individuals, $F \in [0, 1]$ is the DE scale factor parameter, and $v \in X$ is the mutant produced.

The interpretation of Eq. (4) in the search space graph $\mathcal{G}$ is as follows: v is generated by starting from the vertex x_{r_0} and following the arcs indicated by $F \odot (x_{r_1} \ominus x_{r_2})$ which is a sequence of arcs' labels (generators) obtained by truncating a shortest path from x_{r_2} to x_{r_1}. This is in line with what is done by the classical differential mutation equation in the Euclidean space, i.e., generate a mutant v by applying to x_{r_0} the vector corresponding to the truncated segment which connects x_{r_2} to x_{r_1}. Indeed, note that the concept of segment in the Euclidean space is analogous to the concept of shortest path in the graph representing a discrete search space.

2.3 The Search Space of Permutations

The definitions provided in the previous sections are abstract and require implementations for concrete spaces. One of the most investigated search space that verifies the properties of finitely generated groups is the space of permutations [2,25].

The permutations of the set $\{1, \dots, n\}$, together with the usual permutation composition, form the so-called *Symmetric group* $\mathcal{S}(n)$. The identity permutation is $\iota = \langle 1, \dots, n \rangle$. Furthermore, since $\mathcal{S}(n)$ is finite, it is also finitely generated.

One of the most useful generating sets for the permutations is the set of *simple transpositions* $ASW \subset \mathcal{S}(n)$, i.e., particular permutations which algebraically encode the *adjacent swap moves*. Formally,

$$ASW = \{\sigma_i : 1 \leq i < n\}, \tag{5}$$

where the $n - 1$ simple transpositions σ_i are permutations such that

$$\sigma_i(j) = \begin{cases} i + 1 & \text{if } j = i, \\ i & \text{if } j = i + 1, \\ j & \text{otherwise.} \end{cases} \tag{6}$$

Given a generic $\pi \in \mathcal{S}(n)$, the composition $\pi \circ \sigma_i$ swaps the i-th and $(i + 1)$-th items in π. Therefore, using the abstract definitions provided before, a minimal decomposition of the difference between two generic permutations π and ρ corresponds to the shortest sequence of adjacent swap moves which transforms π into ρ.

A minimal decomposition for a generic permutation $\pi \in \mathcal{S}(n)$, in terms of ASW, can be obtained by ordering the items in π by using a sorting algorithm based on adjacent swap moves. The sequence of generators corresponding to the moves performed during the sorting process is annotated, then reversing this sequence produces a minimal decomposition [22].

As widely known, the bubble-sort algorithm sorts any given array by using a minimal number of adjacent swap moves, therefore it can be used for computing a minimal decomposition of any permutation in terms of ASW. Anyway, since there can be more than one minimal decompositions, a randomized variant of bubble-sort, namely *RandBS*, has been proposed in [22].

RandBS exploits the concept of inversion and the property that the identity permutation ι is the only permutation without inversions. Formally, (i, j) is an inversion of a given permutation π if and only if $i < j$ and $\pi(i) > \pi(j)$. Moreover, a permutation with a positive number of inversions has to have at least one *adjacent inversion*, i.e., an inversion of the form $(i, i+1)$. Therefore, *RandBS*(π) decreases the inversions of π by first computing its adjacent inversions and, then, iteratively applying adjacent swaps corresponding to those inversions. At the end of this process π will be transformed into the identity ι, thus the reverse of the sequence of adjacent swaps is a minimal decomposition of π.

RandBS has been proved to have $\Theta(n^2)$ complexity. For further implementation details, proofs of correctness and complexity we refer the interested reader to [22].

3 Permutations with Repetition

3.1 Motivations and Preliminary Definitions

The search space of permutations arises in a variety of combinatorial problems such as, just to name a few: the permutation flowshop scheduling problem, the linear ordering problem, the quadratic assignment problem and the traveling salesman problem. Without loss of generality, an n-length permutation is an ordering of the set $\{1, \ldots, n\}$, thus the items in this ordering are all different from each other.

However, there exist other important combinatorial problems for which it is required that some items can appear several times in the ordering. For instance, in the job shop scheduling problem [11], the items are the jobs to be scheduled and repeated items accommodate the fact that some jobs need to be processed on more than one machine. Repeated items also allow to handle some partitioning problems, such as the balanced multiway graph partitioning problem [17].

We can encode solutions to these problems by means of *permutations with repetition*, i.e., orderings of a given multiset.

A multiset M is a collection of possibly repeated items, the size (or cardinality) of the collection is denoted by $|M|$, and its support *Supp*(M) is the set of all different items appearing in M. For example, the multiset $M = \{1, 1, 2, 2, 3, 3\}$ has cardinality $|M| = 6$ and support *Supp*$(M) = \{1, 2, 3\}$.

Given a multiset M with support $\{1, \ldots, n\}$ and cardinality $q > n$, a permutation with repetition of M is an ordering of the q items in M. We denote by $\mathcal{R}_M$ the set of all the permutations with repetition of M. Considering the multiset M in the previous example, a possible permutation with repetition is $x = \langle 2, 1, 3, 3, 2, 1 \rangle$.

The search space $\mathcal{R}_M$ has size

$$|\mathcal{R}_M| = \frac{q!}{\prod_{i \in Supp(M)} m_M(i)!}, \tag{7}$$

where $m_M(i)$ is the multiplicity of the item i in M, i.e., the number of times i appears in M. Therefore, though $|\mathcal{R}_M| < |\mathcal{S}(q)|$, the size of the search space is anyway exponential with respect to the length of the orderings. This is the main reason of why the combinatorial problems with solutions in $\mathcal{R}_M$ are usually NP-hard.

For the sake of readability, in the rest of the paper, we will use the acronym *PwR* in place of the phrase "permutation with repetition".

3.2 Discrete Operators for Permutations with Repetition

Differently from classical permutations, it is not apparent how to define an internal operation on $\mathcal{R}_M$ which obeys to the group properties. As a consequence, it is not possible to directly use the discrete algebraic operators as defined in Sect. 2.

Anyway, the same simple search moves considered for permutations, i.e., swaps of adjacent items, can be used to move between permutations with repetition. Indeed, all the PwRs in $\mathcal{R}_M$ can be thought as vertices of a search space graph where, as before, its arcs are labelled by adjacent swap moves. Hence, the solutions $x, y \in \mathcal{R}_M$ are neighbors to each other if and only if y can be obtained from x (or vice versa) by swapping two adjacent items in x (or y).

By recalling that the adjacent swap move between the i-th and $(i+1)$-th items (of a normal permutation, but also of a PwR) can be represented as the very simple permutation $\sigma_i \in ASW$ defined in Eq. (6), we have that a path between two given PwRs $x, y \in \mathcal{R}_M$ is a composition of adjacent swaps, i.e., a generic $|M|$-length permutation $\pi \in \mathcal{S}(|M|)$. Clearly, in this space we do not have the dichotomy observed in the Symmetric group $\mathcal{S}(n)$ that is: solutions and paths between solutions have different representations. Solutions are elements of $\mathcal{R}_M$, while paths/moves between solutions are elements of $\mathcal{S}(|M|)$.

The absence of the solution-move dichotomy does not allow to use the same algebraic definitions given in Sect. 2. Nevertheless, we can exploit the graph structure of $\mathcal{R}_M$ in order to derive reasonable definitions for the discrete sum, difference and scalar multiplication operators. These definitions are in line with the geometrical interpretations given in the previous section.

Discrete Sum. The discrete sum operator $\boxplus : \mathcal{R}_M \times \mathcal{S}(|M|) \to \mathcal{R}_M$ which, given a solution $x \in \mathcal{R}_M$ and a move $\pi \in \mathcal{S}(|M|)$, produces the new solution $y = x \boxplus \pi$ by applying to x all the adjacent swap moves appearing in a minimal decomposition of π in terms of ASW.

Discrete Difference. The discrete difference operator $\boxminus : \mathcal{R}_M \times \mathcal{R}_M \to \mathcal{S}(|M|)$ applied to two solutions $x, y \in \mathcal{R}_M$ produces the permutation $\pi = x \boxminus y$ whose minimal decomposition in terms of ASW is formed by the sequence of adjacent swaps that transform y into x. It is interesting to note that, similarly to what happens in the classical Euclidean space, $\boxplus$ and $\boxminus$ are consistent to each other, i.e., for any $x, y \in \mathcal{R}_M$, $x \boxplus (y \boxminus x) = x$.

Discrete Scalar Multiplication. Regarding the scalar multiplication, let observe that practically it is only used to scale-down a move or path in the space[1]. With this regard, see also the geometric interpretation of ADE in Sect. 2.2. Since a move in the search space of PwRs is a normal permutation, we can use unmodified the operator $\odot$ defined in Sect. 2 for the permutation space.

3.3 Implementation of the Discrete Operators

The definition previously given for $\boxplus$ actually indicates also its implementation. As noted in Sect. 2.3, decomposing the permutation costs $\Theta(|M|^2)$. Luckily, given $x \in \mathcal{R}_M$ and $\pi \in \mathcal{S}(|M|)$, it is possible to compute $x \boxplus \pi$ in linear time without decomposing π. Formally, by denoting with $x(i)$ the i-th item of the PwR x, we have that:

$$(x \boxplus \pi)(i) = x(\pi(i)). \tag{8}$$

It is easy to see that applying Eq. (8) to any item $i \in \{1, \ldots, |M|\}$ is equivalent to sequentially applying to x all the adjacent swaps in a decomposition of π.

Let also note that the operator $\boxplus$ is actually a (right) *group action* [18] of the Symmetric group $\mathcal{S}(|M|)$ on the set $\mathcal{R}_M$. Indeed, by using the Polish notation for the sake of readability, it is easy to verify that $\boxplus$ satisfies the two axioms of the (right) group action functions [18]: (i) $\boxplus(x, \iota) = x$ for all $x \in \mathcal{R}_M$, and (ii) $\boxplus(x, \pi \circ \sigma) = \boxplus(\boxplus(x, \pi), \sigma)$ for any $\pi, \sigma \in \mathcal{S}(|M|)$ and $x \in \mathcal{R}_M$.

For the discrete difference $\boxminus$, we first need to define the *canonical PwR* $e \in \mathcal{R}_M$ as the ordering of M whose items are increasingly sorted. For instance, given $M = \{1, 1, 2, 2, 3, 3\}$, its canonical PwR is $e = \langle 1, 1, 2, 2, 3, 3 \rangle$.

Furthermore, let observe that the concept of inversion, introduced in Sect. 2.3, is also defined on the permutations with repetition, and e is the only PwR without inversions.

Therefore, it is possible to use *RandBS* – or, if randomness is not required, any other bubble-sort variant – to sort any PwR x, towards the canonical PwR e, by using an optimal number of adjacent swaps. The optimality derives from the facts that: (i) bubble-sort schemes are known to be optimal when all items are different, and (ii) useless adjacent swaps between equivalent items in a PwR are avoided because the pairs of equivalent items cannot form inversions.

Hence, we are now able to find the sequence of adjacent swaps for moving from any PwR x towards e, i.e., we are able to compute $e \boxminus x$. Moreover, by

[1] Even in the Euclidean space $\mathbb{R}^n$, multiplying a vector by a scalar has a geometric meaning only if we interpret this vector as a proper free vector and not as a point in the space.

observing the commutative diagram depicted in Fig. 1, we can generalize the computation $x \boxminus y$ to any $x, y \in \mathcal{R}_M$. In this diagram, any arrow connects two PwRs and is labelled with the permutation which encodes the sequence of adjacent swaps that transform the tail of the arrow into its head. Equivalently, the label of the arrow is the difference between the head PwR and the tail PwR.

$$\begin{tikzcd}
y \arrow[r, "\pi"] \arrow[d, "\sigma"] & x \arrow[dl, "\rho"] \\
e &
\end{tikzcd}$$

Fig. 1. Commutative diagram showing how to compute $\pi = x \boxminus y$

Since we know how to compute $\sigma = e \boxminus y$ and $\rho = e \boxminus x$, we can now define the difference between two generic PwRs as:

$$x \boxminus y = \pi = \sigma \circ \rho^{-1}. \tag{9}$$

4 Algebraic Differential Evolution for Permutations with Repetition

In this section we define an Algebraic Differential Evolution scheme for Permutations with Repetition (ADE-PR) which is based on the ADE scheme described in Sect. 2.2 and the discrete operators for the PwR representation depicted in Sect. 3.

ADE-PR evolves a population of N permutations with repetition by means of the genetic operators: differential mutation, crossover and selection. Its working scheme, depicted in Algorithm 1, is similar to those of ADE and classical DE. The main difference is that the population of ADE-PR is composed by individuals represented as permutations with repetition.

ADE-PR optimizes a given objective function f defined on the search space $\mathcal{R}_M$. Its control parameters are: the population size N, the scale factor $F \in [0, 1]$ and the crossover strength $CR \in [0, 1]$ (the latter may not be present depending on the chosen crossover operator).

In Algorithm 1, the population is randomly initialized in lines 2–3, then the evolution is performed in the main cycle in lines 4–12 until a given termination criterion is satisfied. For each population individual x_i, a mutant v_i is generated in line 6 by exploiting the differential mutation scheme which is implemented by means of the discrete operators for PwRs previously introduced. Then, a trial PwR u_i is obtained, in line 7, by hybridizing x_i and v_i by means of a chosen

Algorithm 1. Main scheme of ADE-PR

```
1: function ADE-PR($f : \mathcal{R}_M \to \mathbb{R}, N \in \mathbb{N}^+, F \in [0,1], CR \in [0,1]$)
2:     for $i \leftarrow 1, \ldots, N$ do
3:         $x_i \leftarrow$ randomly sample a PwR from $\mathcal{R}_M$
4:     while termination criterion is not satisfied do
5:         for $i \leftarrow 1, \ldots, N$ do
6:             $v_i \leftarrow x_{r_0} \boxplus F \odot (x_{r_1} \boxminus x_{r_2})$
7:             $u_i \leftarrow$ Crossover($x_i, v_i, CR$)
8:             Optionally apply a local search procedure on $u_i$
9:         for $i \leftarrow 1, \ldots, N$ do
10:            if $f(u_i) < f(x_i)$ then
11:                $x_i \leftarrow u_i$
12:        Optionally perform a population restart
13:    return $x_{best}$
14: end function
```

crossover operator working on the PwR representation (like, for instance, GOX, GPMX and PPX [8,9]). In lines 9–10, if u_i is fitter than x_i, it enters the next-generation population by replacing x_i.

Moreover, it is possible to integrate into the ADE-PR scheme: (i) a local search scheme, purposely defined for the problem at hand, in order to refine the search, and (ii) a restart procedure which is often useful in combinatorial problems whose search space is finite.

Note also that the parameters F and CR can be self-adapted during the evolution using one of the many self-adaptive DE schemes in the literature.

The key operator of ADE-PR is the newly introduced differential mutation scheme which directly works with permutations with repetition. The expression in line 6 of Algorithm 1 can be interpreted as follows. The mutant v_i is generated by applying to the PwR x_{r_0} a sequence of adjacent swap moves which is a prefix of the sequence of moves that transform the PwR x_{r_2} into x_{r_1}. Clearly, the length of the prefix is regulated by the scale factor $F \in [0, 1]$. This interpretation is in line with what happens in the continuous DE and for ADE in a search space representable as a finitely generated group.

5 ADE-PR for the Job Shop Scheduling Problem

As a case study, we describe an implementation of ADE-PR for solving the Job Shop Scheduling Problem (JSSP). The resulting algorithm, called ADE-PR-JSSP, follows the scheme depicted in Sect. 4, i.e., it starts with a population of randomly initialized PwRs which are evolved by means of the following operators: discrete differential mutation, GOX crossover [9], selection, local search for the JSSP, and restart procedure. Furthermore, the parameter F used in the discrete differential mutation is self-adapted by means of the jDE method [10], while the GOX crossover, as defined in [9], has no parameter.

In the next subsections we describe: the definition of the JSSP, the procedure for converting a permutation with repetition to a feasible JSSP schedule, the local search operator and the restart scheme adopted in ADE-PR-JSSP.

5.1 Definition of the Problem

The Job Shop Scheduling Problem (JSSP) is an important scheduling problem with many applications in the manufacturing and service industry [12,13].

An instance of the JSSP is defined in terms of a set $\mathcal{J}$ of n jobs $J_1, \ldots, J_n$ and a set $\mathcal{M}$ of m machines $\mu_1, \ldots, \mu_m$. Each job J_i is composed by m operations $O_{i1}, \ldots, O_{im}$. Every operation O_{ij} has a processing time p_{ij} and has to be executed by the machine $\mu_{ij} \in \mathcal{M}$. All the operations within a given job are linearly ordered, while no constraint is defined among operations belonging to different jobs. The set of all the operations is denoted by $\mathcal{O}$.

A feasible schedule s consists in assigning to each operation $O_{ij} \in \mathcal{O}$ a start time s_{ij} such that the following constraints are satisfied: for each $i = 1, \ldots, n$ and $j = 1, \ldots, m-1$,

$$s_{ij} \leq s_{i,j+1}, \tag{10}$$

and, for each $O_{ij}, O_{hk} \in \mathcal{O}$ with $\mu_{ij} = \mu_{hk}$,

$$s_{ij} \geq e_{hk} \text{ or } s_{hk} \geq e_{ij}, \tag{11}$$

where $e_{ij} = s_{ij} + p_{ij}$ is the end time of the operation O_{ij}.

A feasible schedule s is optimal if it optimizes a given objective function. In this paper, the aim is to minimize the makespan

$$C_{max}(s) = \max_{ij} e_{ij}. \tag{12}$$

The JSSP has been approached using a variety of different techniques. In the recent survey [11] many evolutionary and meta-heuristic approaches to solve the JSSP are described: Particle Swarm Optimization, Ant Colony Optimization, Variable Neighborhood Search, Tabu Search, Genetic Algorithms, and several others.

5.2 From a Permutation with Repetition to a JSSP Schedule

The solutions of ADE-PR-JSSP are represented as PwRs over the multiset $M_{m,n}$, whose support is $\{1, \ldots, n\}$, and such that every item in $M_{m,n}$ has multiplicity m. Hence, ADE-PR-JSSP navigates the search space of the permutations with repetition in $\mathcal{R}_{M_{m,n}}$.

This representation, called *operation-based representation* [12], was firstly introduced by [8] and has the important property that it generates only feasible solutions.

The operation-based representation is based on the fact that each operation $O_{ij} \in \mathcal{O}$ can be uniquely identified by the integer number $(i-1)m+j$. Therefore, a PwR $x \in \mathcal{R}_{M_{m,n}}$ is decoded to a JSSP schedule by using a two-phase procedure.

In the first phase, a permutation $\pi_x \in \mathcal{S}(mn)$, representing a total order $\prec_x$ among the operations in $\mathcal{O}$, is built from x as follows.

For each $h = 1, \ldots, mn$, let $j = x(h)$ be the h–th item of x and let k be the number of items x_l, for $1 \leq l \leq h$, such that $x_l = j$, then $\pi_x(h)$ is set to $(j-1)m + k$. Then, $\pi_x(h)$ corresponds to the operation $O_{j,k}$.

It is easy to see that $\prec_x$ respects the constraint (10) by construction. Indeed $O_{ij} \prec_x O_{i,j+1}$, for each pair of indices i and $j < m$. Moreover, for each pair of operations $O_{ij}, O_{hk} \in \mathcal{O}$ assigned to the same machine, $\prec_x$ states in which order the two operations have to be executed.

The second phase assigns to each operation O_{ij} a start time s_{ij} as the maximum among the end time of $O_{i,j-1}$ and the end times of all the operations O_{hk} preceding O_{ij}, with respect to $\prec_x$, and such that $\mu_{ij} = \mu_{hk}$.

The obtained schedule is feasible and respects the precedence relations induced by π_x. Therefore, by calling the conversion procedure as *GenerateSchedule* and given a PwR $x \in \mathcal{R}_{M_{m,n}}$, we have that the fitness of x in ADE-PR-JSSP is the makespan of the corresponding schedule, i.e., $C_{max}\left(\mathit{GenerateSchedule}(x)\right)$.

5.3 Local Search for the JSSP

We have designed ADE-PR-JSSP in such a way that every trial individual $u_i \in \mathcal{R}_{M_{m,n}}$, at every generation of the algorithm, undergoes a local search procedure with probability p_{LS}.

Before applying the local search, the trial individual, represented as a PwR, is first converted to a schedule by means of the procedure described in Sect. 5.2.

The local search is based on the neighborhood $\mathcal{N}^\star$, as described in [21], and works as follows. At each iteration the *critical path* of the current schedule s is computed. This path is the sequence of consecutive operations (where the end time of any operation coincides with the start time of the successive operation in the path) which has the maximum completion time (which corresponds to the makespan of s). Then, the blocks of consecutive operations assigned to the same machine are detected in the critical path. For each block B, two swaps are tried: one exchanges the first two operations in B, while the other exchanges the last two operations. The swap which most reduces the makespan is performed and the schedule is updated accordingly. If no swap produces a better makespan, the local search terminates.

Now, the local optimal schedule is converted back to a PwR and replaces the seed individual u_i in the population of ADE-PR-JSSP. The conversion can be easily implemented by considering a topological sorting in the precedence graph of the local optimal schedule.

After some preliminary experiments, we set the probability p_{LS} to apply the local search as

$$p_{LS}(t) = \frac{t}{T} \cdot p_{LS}^{end} + \left(1 - \frac{t}{T}\right) \cdot p_{LS}^{start}, \tag{13}$$

where t is the current computational time, T is the budget for the execution time, p_{LS}^{start} is the probability of applying the local search at time $t = 0$, and $p_{LS}^{end} >$

p_{LS}^{start} is the probability at time $t = T$. Hence, the local search is progressively applied more often as time passes. This behavior should favor exploration in the earlier phase of the evolution, while exploitation is intensified with the passing of time.

5.4 Restart Scheme

The restart mechanism is implemented by replacing all the population individuals, except the best one, with new randomly generated PwRs.

A restart is performed when the algorithm has not been able to improve its best solution so far after $T \cdot r_{restart}$ seconds, where T is the total allotted running time and $r_{restart} < 1$ is the parameter which regulates how often this operation should be performed at most.

6 Experiments

ADE-PR-JSSP has been experimentally validated on some commonly adopted benchmarks for the JSSP, namely: the *ft*, *la*, and *orb* benchmark suites [15][2]. The benchmarks contain a total of 53 JSSP instances with $n \cdot m$ ranging from 36 to 300.

After some preliminary experiments, the population size has been set to $N = 25$ individuals, the range for the application probability of the local search has been set using $p_{LS}^{start} = 0$ and $p_{LS}^{end} = 1$, while the restart parameter $r_{restart}$ has been set to 0.1.

The executions of ADE-PR-JSS have been carried out on a machine equipped with the Intel Xeon CPU E5-2620 v4 clocking at 2.10 GHz. Every execution terminates after a time budget of $T = 4mn$ seconds has been exhausted. Moreover, $R = 15$ executions per instance have been run.

The presentation of the experimental results is divided in three groups, according to the values of mn: Table 1 refers to the instances with $nm < 100$, Table 2 to those with $nm = 100$, and Table 3 to the remaining instances. In these three tables we present, for each instance: the sizes n and m, the average (Avg_i) and minimum (Min_i) fitness values obtained by ADE-PR-JSSP in the R runs, the known optimal value (Opt_i) for the instance (taken from the recently published survey paper [15]), and the average relative percentage deviation computed as $ARPD_i = 100 \times \frac{Avg_i - Opt_i}{Opt_i}$. Moreover, the minimum Min_i is reported in boldface when it matches the known optimal value Opt_i.

Interestingly, in 38 out of 53 instances, ADE-PR-JSSP reached the optimal value at least once, while, in 22 instances, this happened in all the executions.

In particular, the results provided in Table 1 refer to small JSSP instances, where ADE-PR-JSSP has been always able to find the optimal value, and for 7 of such instances this happened in all the executions. Indeed, the average ARPD for this set of instances is rather small, i.e., 0.167%.

[2] These JSSP instances can be downloaded from the website http://jobshop.jjvh.nl.

Table 1. Experimental results on instances with $nm < 100$

Instance	n	m	Avg	ARPD	Min	Opt
ft06	6	6	55.000	0.000	**55**	55
la01	10	5	666.000	0.000	**666**	666
la02	10	5	660.133	0.784	**655**	655
la03	10	5	602.867	0.983	**597**	597
la04	10	5	590.400	0.068	**590**	590
la05	10	5	593.000	0.000	**593**	593
la06	15	5	926.000	0.000	**926**	926
la07	15	5	890.000	0.000	**890**	890
la08	15	5	863.000	0.000	**863**	863
la09	15	5	951.000	0.000	**951**	951
la10	15	5	958.000	0.000	**958**	958

Table 2. Experimental results on instances with $nm = 100$

Instance	n	m	Avg	ARPD	Min	Opt
ft10	10	10	937.933	0.853	**930**	930
ft20	20	5	1174.600	0.824	**1165**	1165
la11	20	5	1222.000	0.000	**1222**	1222
la12	20	5	1039.000	0.000	**1039**	1039
la13	20	5	1150.000	0.000	**1150**	1150
la14	20	5	1292.000	0.000	**1292**	1292
la15	20	5	1207.000	0.000	**1207**	1207
la16	10	10	949.800	0.508	**945**	945
la17	10	10	785.267	0.162	**784**	784
la18	10	10	848.667	0.079	**848**	848
la19	10	10	845.667	0.435	**842**	842
la20	10	10	906.667	0.517	**902**	902
orb01	10	10	1078.267	1.819	1064	1059
orb02	10	10	890.800	0.315	889	888
orb03	10	10	1019.000	1.393	**1005**	1005
orb04	10	10	1018.267	1.320	1011	1005
orb05	10	10	891.400	0.496	889	887
orb06	10	10	1025.800	1.564	1021	1010
orb07	10	10	401.667	1.175	**397**	397
orb08	10	10	906.800	0.868	**899**	899
orb09	10	10	943.467	1.014	**934**	934
orb10	10	10	944.000	0.000	**944**	944

Table 3. Experimental results on instances with $nm > 100$

Instance	n	m	Avg	ARPD	Min	Opt
la21	15	10	1059.933	1.332	1047	1046
la22	15	10	930.133	0.338	**927**	927
la23	15	10	1032.000	0.000	**1032**	1032
la24	15	10	941.667	0.713	938	935
la25	15	10	982.867	0.600	982	977
la26	20	10	1218.000	0.000	**1218**	1218
la27	20	10	1257.533	1.825	1242	1235
la28	20	10	1221.533	0.455	**1216**	1216
la29	20	10	1190.067	3.304	1174	1152
la30	20	10	1355.000	0.000	**1355**	1355
la31	30	10	1784.000	0.000	**1784**	1784
la32	30	10	1850.000	0.000	**1850**	1850
la33	30	10	1719.000	0.000	**1719**	1719
la34	30	10	1721.000	0.000	**1721**	1721
la35	30	10	1888.000	0.000	**1888**	1888
la36	15	15	1286.267	1.441	1278	1268
la37	15	15	1435.600	2.763	1418	1397
la38	15	15	1210.867	1.243	1202	1196
la39	15	15	1249.400	1.330	1246	1233
la40	15	15	1240.133	1.484	1228	1222

Table 2 shows that on the 22 selected instances with $nm = 100$ the algorithm has been able to find the optimal value: at least once on 17 instances, and in all the executions in 6 cases. As expected, the average ARPD for this second set of instances, 0.606%, is larger than the previous, but anyway close to 0.

In Table 3 it is possible to see that ADE-PR-JSSP reached the known optimal value: at least once in half the instances (10 out of 20), and in all the executions for 8 of them. Therefore, the average ARPD for this last set of instances is slightly larger than the other: 0.841%. Moreover, Table 3 also shows that the instances with $m = 15$ are much harder and the average ARPD restricted to this subset raises to 1.652%.

Summarizing, the overall ARPD obtained by averaging on all the 53 instances is 0.604, thus promoting the proposed approach as a method competitive with respect to the known values for the considered benchmarks.

7 Conclusion and Future Work

In this paper, we have extended the algebraic framework for evolutionary computation previously proposed in [22] in order to handle the search space of permutations with repetition. The newly proposed discrete operators allowed to design an Algebraic Differential Evolution called ADE-PR which can be applied to any combinatorial optimization problem whose solutions may be represented as permutations of possibly repeated items.

In particular, ADE-PR has been devised for the Job Shop Scheduling Problem (JSSP). In order to validate the effectiveness of the proposed approach, experiments have been on a set of widely adopted benchmark instances for the JSSP. The experimental results show that our proposal is competitive with respect to the known optimal objective values for the considered benchmarks.

Possible future lines of research are: apply ADE-PR to partitioning problems; use simple search moves other than the swaps of adjacent items; design other algebraic evolutionary algorithms, like the APSO [23], in order to work with permutations with repetition; and generalize the approach to other search spaces, by means of the algebraic concept of *group action*, in order to see the deployed discrete operators as projections from a known space which can be represented as a group to other more general combinatorial spaces.

Acknowledgement. The research described in this work has been partially supported by: "Università per Stranieri di Perugia – Finanziamento per Progetti di Ricerca di Ateneo –PRA 2020", and by RCB-2015 Project "Algoritmi Randomizzati per l'Ottimizzazione e la Navigazione di Reti Semantiche" and RCB-2015 Project "Algoritmi evolutivi per problemi di ottimizzazione combinatorica" of Department of Mathematics and Computer Science of University of Perugia.

References

1. Baioletti, M., Milani, A., Santucci, V.: Algebraic crossover operators for permutations. In: 2018 IEEE Congress on Evolutionary Computation (CEC 2018), pp. 1–8 (2018). https://doi.org/10.1109/CEC.2018.8477867
2. 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
3. Baioletti, M., Milani, A., Santucci, V.: Automatic algebraic evolutionary algorithms. In: Pelillo, M., Poli, I., Roli, A., Serra, R., Slanzi, D., Villani, M. (eds.) WIVACE 2017. CCIS, vol. 830, pp. 271–283. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-78658-2_20
4. Baioletti, M., Milani, A., Santucci, V.: Learning Bayesian networks with algebraic differential evolution. In: Auger, A., Fonseca, C.M., Lourenço, N., Machado, P., Paquete, L., Whitley, D. (eds.) PPSN 2018. LNCS, vol. 11102, pp. 436–448. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-99259-4_35
5. Baioletti, M., Milani, A., Santucci, V.: MOEA/DEP: an algebraic decomposition-based evolutionary algorithm for the multiobjective permutation flowshop scheduling problem. In: Liefooghe, A., López-Ibáñez, M. (eds.) EvoCOP 2018. LNCS, vol. 10782, pp. 132–145. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-77449-7_9

6. Baioletti, M., Milani, A., Santucci, V.: Variable neighborhood algebraic differential evolution: an application to the linear ordering problem with cumulative costs. Inf. Sci. **507**, 37–52 (2020). https://doi.org/10.1016/j.ins.2019.08.016
7. Baioletti, M., Milani, A., Santucci, V., Bartoccini, U.: An experimental comparison of algebraic differential evolution using different generating sets. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, pp. 1527–534 (2019). https://doi.org/10.1145/3319619.3326854
8. Bierwirth, C.: A generalized permutation approach to job shop scheduling with genetic algorithms. Oper. Res. Spektrum **17**(2), 87–92 (1995)
9. Bierwirth, C., Mattfeld, D.C., Kopfer, H.: On permutation representations for scheduling problems. In: Voigt, H.-M., Ebeling, W., Rechenberg, I., Schwefel, H.-P. (eds.) PPSN 1996. LNCS, vol. 1141, pp. 310–318. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61723-X_995
10. Brest, J., Greiner, S., Boskovic, B., Mernik, M., Zumer, V.: Self-adapting control parameters in differential evolution: a comparative study on numerical benchmark problems. IEEE Trans. Evol. Comput. **10**(6), 646–657 (2006)
11. Çaliş, B., Bulkan, S.: A research survey: review of AI solution strategies of job shop scheduling problem. J. Intell. Manuf. **26**(5), 961–973 (2015)
12. Cheng, R., Gen, M., Tsujimura, Y.: A tutorial survey of job-shop scheduling problems using genetic algorithms, Part I: representation. Comput. Ind. Eng. **30**(4), 983–997 (1996)
13. Cheng, R., Gen, M., Tsujimura, Y.: A tutorial survey of job-shop scheduling problems using genetic algorithms, Part II: hybrid genetic search strategies. Comput. Ind. Eng. **36**(2), 343–364 (1999)
14. González, M.Á., Oddi, A., Rasconi, R.: Multi-objective optimization in a job shop with energy costs through hybrid evolutionary techniques. In: Proceedings of the 27th International Conference on Automated Planning and Scheduling (2017)
15. van Hoorn, J.J.: The current state of bounds on benchmark instances of the job-shop scheduling problem. J. Sched. **21**(1), 127–128 (2018)
16. Jovanovski, J., Arsov, N., Stevanoska, E., Siljanoska Simons, M., Velinov, G.: A meta-heuristic approach for RLE compression in a column store table. Soft Comput. **23**(12), 4255–4276 (2019)
17. Kim, J., Hwang, I., Kim, Y.H., Moon, B.R.: Genetic approaches for graph partitioning: a survey. In: Proceedings of the GECCO 2011, pp. 473–480. ACM (2011)
18. Lang, S.: Algebra, vol. 211. Springer, Heidelberg (2002). https://doi.org/10.1007/978-1-4613-0041-0
19. Milani, A., Santucci, V.: Asynchronous differential evolution. In: Proceedings of 2010 IEEE Congress on Evolutionary Computation (CEC 2010), pp. 1–7 (2010). https://doi.org/10.1109/CEC.2010.5586107
20. Moraglio, A., Kim, Y.H., Yoon, Y., Moon, B.R.: Geometric crossovers for multiway graph partitioning. Evol. Comput. **15**(4), 445–474 (2007)
21. Nowicki, E., Smutnicki, C.: An advanced tabu search algorithm for the job shop problem. J. Sched. **8**(2), 145–159 (2005)
22. 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
23. Santucci, V., Baioletti, M., Milani, A.: Tackling permutation-based optimization problems with an algebraic particle swarm optimization algorithm. Fundamenta Informaticae **167**(1–2), 133–158 (2019). https://doi.org/10.3233/FI-2019-1812

24. Santucci, V., Baioletti, M., Di Bari, G., Milani, A.: A binary algebraic differential evolution for the multidimensional two-way number partitioning problem. In: Liefooghe, A., Paquete, L. (eds.) EvoCOP 2019. LNCS, vol. 11452, pp. 17–32. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-16711-0_2
25. Santucci, V., Ceberio, J.: Using pairwise precedences for solving the linear ordering problem. Appl. Soft Comput. **87** (2020). https://doi.org/10.1016/j.asoc.2019.105998
26. Santucci, V., Milani, A.: Particle swarm optimization in the EDAs framework. In: Gaspar-Cunha, A., Takahashi, R., Schaefer, G., Costa, L. (eds.) Soft Computing in Industrial Applications, pp. 87–96. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20505-7_7
27. Sörensen, K.: Distance measures based on the edit distance for permutation-type representations. J. Heuristics **13**(1), 35–47 (2007)