Variable Neighborhood Algebraic Differential Evolution: an application to the Linear Ordering Problem with Cumulative Costs

Marco Baioletti[a,*], Alfredo Milani[a], Valentino Santucci[b]

[a] *Department of Mathematics and Computer Science, University of Perugia, Italy*
[b] *Department of Humanities and Social Sciences, University for Foreigners of Perugia, Italy*

Abstract

Algebraic variants of the Differential Evolution (DE) algorithm have been recently proposed to tackle permutation-based optimization problems by means of an algebraic framework, which allows to directly encode the solutions as permutations. The algebraic DE in the permutation space can be characterized by considering different neighborhood definitions such as swapping two adjacent items, swapping any two items, shifting an item to a given position. Here we propose the Variable Neighborhood Differential Evolution for Permutations (VNDEP), which adaptively searches the three neighborhoods together based on a method of dynamic reward. We provide an extensive and systematic analysis of the theoretical tools required in VNDEP, by studying the complexity of the proposed algorithmic components and by introducing the possibility to use a scale factor parameter larger than one. Experiments have been held on a widely used benchmark suite for the Linear Ordering Problem with Cumulative Costs, where VNDEP has been compared with four known permutation-based DE schemes and with respect to the state-of-the-art results for the considered instances. The experiments clearly show that VNDEP systematically outperforms the competitor algorithms and, most impressively, 32 new best known solutions, of the 50 most challenging instances, have been obtained.

Keywords: Discrete Differential Evolution; Algebraic Differential Evolution; Adaptive Differential Evolution; Linear Ordering Problem with Cumulative Costs; Variable Neighborhood Search

1. Introduction

In this work we propose a Variable Neighborhood Algebraic Differential Evolution for Permutation problems (VNDEP) by extending the DEP algorithm

*All the authors contributed equally. Corresponding author:
Email address: marco.baioletti@unipg.it (Marco Baioletti)

preliminary described in [**?**], which in turn is based on the algebraic framework introduced in [**?**].

In this framework, the neighborhood relations among the permutations are related to the concept of *generating set*, i.e., a small subset of permutations – called *generators* – that, when applied to a permutation x, allow to find all the neighbors of x. Different generating sets (thus neighborhoods) are possible for the permutation space: in [**?**] the adjacent swap moves have been considered, while in [**?**] exchange and insertion moves have been used.

In [**?**] DEP has obtained some new best known solutions on the Linear Ordering Problem with Cumulative Costs (LOPCC). These results have been obtained by using multiple variants of DEP which differ from each other by the generating set adopted. Therefore, here we are interested in investigating whether the choice of the generating set can be automated and adapted to the problem at hand.

The basic idea of VNDEP is to use the three generating sets together, in a single execution of the algorithm, by adaptively selecting them using a strategy based on rewards assigned to the most successful generating sets during the evolution. The rewards assignment has been designed to take into account the behavior usually observed in evolutionary algorithms, where it is easy to produce an improvement in the early iterations, while it gets progressively more difficult with the passing of the evolution.

Furthermore, the previous work [**?**] has a limitation related to the range of values allowed for the scale factor parameter. Here, we remove this limitation by introducing, for all the three generating sets, the possibility of multiplying a permutation by any non-negative real number. Hence, we provide a formal and precise description of VNDEP and we systematize and extend the concepts in [**?**] by presenting an analysis of the theoretical tools and algorithmic components required by the algebraic framework.

As a case of study, VNDEP has been experimented with the Linear Ordering Problem with Cumulative Costs (LOPCC). The LOPCC is an NP-hard problem introduced in [**?**] as a cumulative variant of the LOP (Linear Ordering Problem [**?**]) with important applications in the design of the mobile-phone telecommunication systems. Given a complete digraph of n nodes with node weights $d_i \geqslant 0$ and arc weights $c_{ij} \geqslant 0$, the LOPCC aims to find a permutation x of the nodes that minimizes the objective function

$$f(x) = \sum_{i=1}^{n} \alpha_{x(i)} \tag{1}$$

where the α-costs are recursively calculated as

$$\alpha_{x(i)} = d_{x(i)} + \textstyle\sum_{j=i+1}^{n} c_{x(i)x(j)}\alpha_{x(j)} \qquad \text{for } i = n, n-1, \ldots, 1. \tag{2}$$

VNDEP has been experimentally compared with respect to the state-of-the-art results for the considered benchmark suite and with four variations of DE based on the random-key technique [**?**], which is the most used method to deal

with permutation problems by means of DE. Additional experiments have been also conducted on widely used benchmark instances of the standard version of the linear ordering problem.

The rest of the paper is organized as follows. Section 2 describes classical DE, its adaptive extensions, and the discrete DE variants for permutations available in literature. Section 3 introduces the algebraic interpretation of the permutations search space. The definitions of the vector-like operations are provided in Section 4, while the proposed algorithmic components are analyzed in Section 5. VNDEP and its adaptive mechanism is then introduced in Section 6. The experimental results are provided and discussed in Section 7, while Section 8 concludes the paper by also providing future lines of research.

2. Related Work

2.1. Classical Differential Evolution

Differential Evolution (DE) [**?**] is a simple and powerful evolutionary algorithm for optimizing non-linear and even non-differentiable functions of the form $f : \mathbb{R}^n \to \mathbb{R}$. DE evolves a population of N real-valued vectors $x_1, \ldots, x_N \in \mathbb{R}^n$ by iteratively applying the three genetic operators: differential mutation, crossover, and selection.

The differential mutation generates a *mutant* y_i for each individual x_i. Several mutation schemes have been proposed [**?** **?**]. The original one is denoted by *rand/1* and it is computed as

$$y_i = x_{r_1} + F \cdot (x_{r_2} - x_{r_3}), \tag{3}$$

where r_1, r_2, r_3 are three random integers in $\{1, \ldots, N\}$ mutually different among them and with respect to i, while $F > 0$ is the DE *scale factor* parameter.

Then, x_i and y_i undergo a crossover operator which produces the offspring z_i. Many crossover schemes have been proposed, see for example [**?** **?**]. Usually the crossover operator is regulated by the parameter $CR \in [0, 1]$.

The new population is then formed by means of the selection operator. The most used is the 1-to-1 selection, i.e., each offspring z_i replaces the corresponding population individual x_i if z_i is fitter than x_i.

For further discussions about the design issues of DE, we refer the interested reader to the two recent review papers [**?**] and [**?**].

The DE parameters F and CR have a great impact on the evolution, hence there have been many proposals of adaptive DE schemes [**?** **?** **?** **?** **?**].

One of the most popular method is jDE [**?**], where every population individual x_i has its own values F_i and CR_i. The offspring z_i inherits F_i from x_i with probability 0.9, otherwise a new value is randomly sampled from $[0.1, 1]$. Analogously, CR_i is inherited with probability 0.9, otherwise it is randomly sampled in $[0, 1]$. If z_i enters the new generation, it keeps its parameter values.

Another self-adaptive method is JADE [**?**], where F is sampled from a Cauchy distribution and CR is sampled from a normal distribution. These two distributions are centered on, respectively, the Lehmer and arithmetic means of

the F and CR values that allowed the offsprings, produced so far, to pass the selection phase. JADE also introduces the mutation scheme current-to-$p$best, which is defined as

$$y_i = x_i + F \cdot (x_{pbest} - x_i) + F \cdot (x_{r_1} - x_{r_2}), \tag{4}$$

where x_{pbest} is randomly chosen among the best $\lceil p \cdot N \rceil$ individuals, while x_{r_1} is selected from the DE population, and x_{r_2} is chosen from a set composed by the population and an external archive $\mathcal{A}$, of size N, containing the most recently replaced individuals.

JADE has been successfully extended by SHADE [**?**], which samples the DE parameters from multimodal mixture distributions. Finally, further variants have been proposed in [**?**] and [**?**].

2.2. Discrete Differential Evolution schemes for Permutations

Many meta-heuristics have been proposed for combinatorial optimization problems (see for example [**? ? ? ? ? ?**]). Here, we briefly review the main applications of DE to permutation problems.

There exist DE schemes specifically tailored for a particular permutation problem which use purposely defined ad-hoc operations, like for instance the DE proposed in [**?**] for the generalized traveling salesman problem. However, these schemes do not easily generalize to other problems. Hence, in this article we consider the most general class of the DE schemes based on the random-key decoders [**? ? ? ?**].

The first random key decoder has been proposed in [**?**] in the context of genetic algorithms. Formally, by denoting with $\mathcal{S}_n$ the set of permutations of the integers in $\{1, \ldots, n\}$, [**?**] introduces a decoder function $RK : \mathbb{R}^n \to \mathcal{S}_n$ which can be used by DE to optimize the objective function $f : \mathcal{S}_n \to \mathbb{R}$ of the permutation problem at hand. Therefore, the only modification to the classical DE is to consider $f(RK(x))$ as the fitness of $x \in \mathbb{R}^n$.

The decoder RK transforms $x \in \mathbb{R}^n$ to the permutation $\pi \in \mathcal{S}_n$ such that the sequence $x_{\pi(1)}, \ldots, x_{\pi(n)}$ is increasingly ordered. For example, given $x = (0.46, 0.91, 0.33, 0.75, 0.51)$, its corresponding permutation is $\pi = RK(x) = \langle 3, 1, 5, 4, 2 \rangle$. Therefore, RK requires to sort the component indexes of x with respect to their corresponding values.

Also a simple variant of RK has been considered in literature [**? ?**]. In this variant, a vector $x \in \mathbb{R}^n$ is decoded to the permutation $\rho \in \mathcal{S}_n$ such that $\rho(i) = r_i$, where r_i is the rank of x_i among the vector components $x_1, \ldots, x_n$ sorted in ascending order. Using the numeric vector of the previous example, $\rho = \langle 2, 5, 1, 4, 3 \rangle$. It is easy to see that the permutation ρ is the inverse permutation of π, i.e., $\rho = \pi^{-1}$, and vice versa. Hence, though not explicitly reported in literature, this decoding scheme, to which we refer with RKI, can be obtained by inverting the result of RK, i.e., $RKI(x) = (RK(x))^{-1}$.

2.3. DE schemes based on algebraic principles

Few DE variants based on algebraic principles have been proposed in literature.

A general algebraic framework has been introduced in [**?**] and [**?**], where the algebraic DE has been applied to the permutation flowshop scheduling problem (PFSP). This algorithm constrains the scale factor in the interval $[0, 1]$ and uses adjacent swap moves to search in the permutation space of the PFSP. The same algorithm has also been applied to the linear ordering problem (using the standard objective function formulation) in [**?**] and [**?**]. Algebraic DEs which use exchange and insertion moves for the permutation space have been introduced in [**?**] for solving the linear ordering problem with cumulative costs. Moreover, a multiobjective variant has been applied in [**?**] to the multiobjective PFSP, while the same algebraic framework has been adopted in [**?**] to search in the space of directed acyclic graphs for learning the structure of a Bayesian network. Finally, a binary variant of the algebraic DE has been introduced in [**?**] and applied to the multidimensional number partitioning problem.

Following a different line of research, a group-theory based DE has been recently proposed in [**?**] with application to knapsack problems.

3. Algebraic Interpretation of the Permutations Search Space

In many combinatorial optimization problems, the set of discrete solutions X is naturally endowed with a composition operator, i.e., there exists a binary operation $\star$ such that, given two solutions $x, y \in X$, $x \star y$ is again a valid solution. Often, X and $\star$ satisfy the group properties [**?** **?**]. Though a variety of discrete search spaces of practical interest, like for instance the spaces of bit-strings and integer vectors [**?**], fall into this category, in this article we focus our attention on the space of permutations.

The set $\mathcal{S}_n$ of the $n!$ permutations of $[n] = \{1, 2, \ldots, n\}$ forms a group with respect to the permutation composition $\circ$: given $x, y \in \mathcal{S}_n$, their composition $x \circ y$ is defined as the permutation $(x \circ y)(i) = x(y(i))$, for all the indexes $i \in [n]$.

The group structure allows to characterize the geometry of the search space and to describe the search moves.

The rest of this section is devoted to introducing the mathematical concepts used later on.

3.1. The Symmetric Group and its Generating Sets

$\mathcal{S}_n$ and the composition operator $\circ : \mathcal{S}_n \times \mathcal{S}_n \to \mathcal{S}_n$ form the so-called *symmetric group*. We denote by $e = \langle 1, 2, \ldots, n \rangle$ the identity permutation, which is the neutral element of $\mathcal{S}_n$, and by $x^{-1} \in \mathcal{S}_n$ the inverse of any $x \in \mathcal{S}_n$.

There exist different subsets $H \subseteq \mathcal{S}_n$ such that any $x \in \mathcal{S}_n$ can be written as a composition of the permutations in H, i.e., $x = h_1 \circ h_2 \circ \cdots \circ h_l$ for some $h_1, h_2, \ldots, h_l \in H$. Hence, $\mathcal{S}_n$ is said to be finitely generated by the generating set H, while the sequence $\langle h_1, h_2, \ldots, h_l \rangle$ is a decomposition of x, and the permutations in H are called generators.

In this article we focus on three generating sets of $\mathcal{S}_n$, denoted by ASW, EXC, and INS, which are particularly useful because they encode in the algebraic language the elementary moves usually considered in the permutations space (see for example [**?** **?**]). Their definitions and properties are as follows.

- $ASW = \{\sigma_i : 1 \leqslant i < n\}$, where σ_i is the identity permutation with the items i and $i+1$ exchanged (e.g., in $\mathcal{S}_5$, $\sigma_2 = \langle 13245 \rangle$). These generators correspond to the *adjacent swap* moves because, given $x \in \mathcal{S}_n$, the permutation $x \circ \sigma_i$ is obtained from x by swapping the adjacent items at positions i and $i+1$.

- $EXC = \{\epsilon_{ij} : 1 \leqslant i < j \leqslant n\}$, where ϵ_{ij} is the identity permutation with the items i and j exchanged (e.g., in $\mathcal{S}_5$, $\epsilon_{14} = \langle 42315 \rangle$). These generators correspond to the *exchange* moves because, the permutation $x \circ \epsilon_{ij}$ is obtained from x by exchanging the items at positions i and j.

- $INS = \{\iota_{ij} : 1 \leqslant i \neq j \leqslant n\}$, where ι_{ij} is the identity permutation with the item i shifted to position j (e.g., in $\mathcal{S}_5$, $\iota_{14} = \langle 23415 \rangle$). These generators model the *insertion* moves because, the permutation $x \circ \iota_{ij}$ is obtained from x by shifting the i–th item to position j.

Their cardinalities are: $|ASW| = n-1$, $|EXC| = \binom{n}{2}$, and $|INS| = (n-1)^2$. All the generating sets are closed with respect to inversion because $\sigma_i^{-1} = \sigma_i$, $\epsilon_{ij}^{-1} = \epsilon_{ji}$, and $\iota_{ij}^{-1} = \iota_{ji}$. Moreover, $ASW \subseteq EXC$ and $ASW \subseteq INS$ hold. For the sake of clarity, in the rest of the paper, we denote with $\mathcal{H}$ the set $\{ASW, EXC, INS\}$.

Given a generating set $H \in \mathcal{H}$, a decomposition $\langle h_1, h_2, \ldots, h_l \rangle$ of any $x \in \mathcal{S}_n$ is minimal if, for any other decomposition $\langle h'_1, h'_2, \ldots, h'_m \rangle$ of x, we have $l \leqslant m$. Although minimal decompositions are not unique in general, they allows to define the weight $|x|$ as the length l of the minimal decompositions of x in terms of H.

Independently from the generating set, the minimal-weight permutation is always the identity e whose weight is $|e| = 0$. Furthermore, since $\mathcal{S}_n$ is finite, for any possible generating set H there exists a non-empty set Ω_H of maximal-weight permutations.

Finally, for any $H \in \mathcal{H}$, minimal decompositions allow also to define a partial order on $\mathcal{S}_n$. Given $x, y \in \mathcal{S}_n$, we write $x \sqsubseteq y$ if, for each minimal decomposition s_x of x, there exists a minimal decomposition s_y of y such that s_x is a prefix of s_y.

3.2. The Cayley Graph of Permutations

The triple $(\mathcal{S}_n, \circ, H)$, with $H \in \mathcal{H}$, can be geometrically associated to the *Cayley graph* $\mathcal{C}(\mathcal{S}_n, \circ, H)$, i.e., the labeled digraph whose vertices are all the permutations in $\mathcal{S}_n$ and there is an arc from x to y labeled by $h \in H$ if and only if $y = x \circ h$.

It is easy to prove that, for any possible sequence of generators s and for any permutation $x \in \mathcal{S}_n$, $\mathcal{C}(\mathcal{S}_n, \circ, H)$ has exactly one path which starts from the vertex x and whose arcs are labeled according to s. Moreover, for all $x \in \mathcal{S}_n$, each directed path from the identity e to x corresponds to a decomposition of x, i.e., if the arc labels occurring in the path are $\langle h_1, h_2, \ldots, h_l \rangle$, then $x = h_1 \circ h_2 \circ \cdots \circ h_l$. As a consequence, shortest paths from e to x correspond to

minimal decompositions of x, i.e., if $e \xrightarrow{h_1} x_1 \xrightarrow{h_2} x_2 \xrightarrow{h_3} \cdots \xrightarrow{h_l} x_l$ is one of the shortest paths in $\mathcal{C}(\mathcal{S}_n, \circ, H)$, then, for any integer $i \in [1, l]$, $\langle h_1, \ldots, h_i \rangle$ is a minimal decomposition of x_i and $|x_i| = i$. Moreover, given $x, y \in \mathcal{S}_n$, $x \sqsubseteq y$ if and only if there exists at least one shortest path from e to y passing by x.

More generally, for all $x, y \in \mathcal{S}_n$, any path from x to y in $\mathcal{C}(\mathcal{S}_n, \circ, H)$ has an algebraic interpretation: if the arc labels in the path are $\langle h_1, h_2, \ldots, h_l \rangle$, then $x \circ (h_1 \circ h_2 \circ \cdots \circ h_l) = y$. Hence, $\langle h_1, h_2, \ldots, h_l \rangle$ is a decomposition of $x^{-1} \circ y$. In particular, shortest paths correspond to minimal decompositions.

Furthermore, the diameter D of $\mathcal{C}(\mathcal{S}_n, \circ, H)$ is equal to the maximum weight, i.e., $D = |\omega|$ for any $\omega \in \Omega_H$. The diameter for ASW is $\binom{n}{2}$, while it is $n - 1$ for both EXC and INS [**?**].

4. Vector-like Operations on Permutations

In this section, we present the vector-like operations $\oplus$, $\ominus$, and $\odot$ that simulate in a meaningful way their numerical counterparts. This, in turn, will allow to consistently redefine the mutation operator of Differential Evolution to directly work with permutations.

The key observation is the dichotomous interpretation of a permutation. From Section 3, any $x \in \mathcal{S}_n$ can be decomposed and seen as a sequence of generators, hence x corresponds to a sequence of arc labels in several paths of the Cayley graph. This observation is crucial, because the permutations can be seen both as *points*, i.e., vertexes in the Cayley graph, and as *vectors*[1], i.e., sequences of generators in shortest paths of the Cayley graph.

4.1. Addition and Subtraction for Permutations

The addition $z = x \oplus y$ is defined as the application of the vector $y \in \mathcal{S}_n$ to the point $x \in \mathcal{S}_n$. Hence, z can be 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 arc labels are $\langle h_1, h_2, \ldots, h_l \rangle$, i.e., $z = x \circ (h_1 \circ h_2 \circ \cdots \circ h_l)$. Since $h_1 \circ h_2 \circ \cdots \circ h_l = y$, the addition $\oplus$ is independent from the generating set and is uniquely defined as

$$x \oplus y := x \circ y. \tag{5}$$

Continuing the analogy with the Euclidean space, the difference between two points is a vector. Given $x, y \in \mathcal{S}_n$, 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 $h_1 \circ h_2 \circ \cdots \circ h_l = x^{-1} \circ y$, we can replace the sequence of labels with its composition, thus making the difference independent from the generating set. Therefore, $\ominus$ is uniquely defined as

$$y \ominus x := x^{-1} \circ y. \tag{6}$$

Though not commutative, both $\oplus$ and $\ominus$, like their numerical counterparts, are consistent with each other, i.e., $x \oplus (y \ominus x) = y$ for all $x, y \in \mathcal{S}_n$. Moreover, the identity e is their neutral element.

[1] With "vector" we intend "free vector", i.e., a vector without point of application.

4.2. Scalar Multiplication for Permutations

Again, as in the Euclidean space, it is possible to multiply a vector by a non-negative scalar in order to stretch its length.

Given $a \geqslant 0$ and $x \in \mathcal{S}_n$, we denote their multiplication with $a \odot x$ and we first identify the conditions that the permutation $a \odot x$ 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 $a \geqslant 1$, $x \sqsubseteq a \odot x$.

Clearly, the scalar multiplication of $\mathbb{R}^n$ satisfies a 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 $a \odot x$ scales, respectively, down or up the permutation x.

It is important to note that, fixed a and x, there may be more than one result satisfying (C1–C3). This is a clear consequence of the non-uniqueness of minimal decompositions. Therefore, we denote with $a \odot x$ a randomly selected permutation satisfying (C1–C3).

Note also that the diameter D induces an upper bound on the possible values for the scalar a. For any $x \in \mathcal{S}_n$, let $\overline{a}_x = \frac{D}{|x|}$, if $a > \overline{a}_x$, (C1) would imply $|a \odot x| > D$ that is impossible. Therefore, we define

$$a \odot x := \overline{a}_x \odot x, \quad \text{when } a > \overline{a}_x. \tag{7}$$

For the sake of clarity, we separately define the operation $a \odot x$ for $a \in [0, 1]$ and for $a > 1$. Both cases employ an abstract procedure which returns a randomly selected minimal decomposition of the permutation in input.

When $a \in [0, 1]$, $a \odot x$ is computed by taking a minimal decomposition of x, truncating it after $\lceil a \cdot |x| \rceil$ generators, and composing the truncated sequence. Therefore, both (C1) and (C2) are satisfied. Moreover, when $a = 1$, $a \odot x = x$. This satisfies both (C2) and (C3). Note also that, due to the ceiling in condition (C1) and similarly to the Euclidean case, $a = 0$ is the unique scalar such that $a \odot x = e$ for all $x \in \mathcal{S}_n$.

When $a > 1$, the condition (C3) can be satisfied if and only if, for the chosen generating set H, there exists a permutation $\omega \in \Omega_H$ such that $x \sqsubseteq \omega$. In this case, by also letting $l = |x|$, there exists a shortest path from e to ω passing by x such as $e \xrightarrow{h_1} \cdots \xrightarrow{h_l} x_l \xrightarrow{h_{l+1}} \cdots \xrightarrow{h_k} x_k \xrightarrow{h_{k+1}} \cdots \xrightarrow{h_D} \omega$, where $x_l = x$. Hence, for the Cayley graph properties, (C1) and (C3) are satisfied by setting $a \odot x = x_k$, with $k = \lceil a \cdot l \rceil$. Anyway, it is possible to make the computation more efficient by exploiting that $a \odot x = (h_1 \circ \cdots \circ h_l) \circ (h_{l+1} \circ \cdots \circ h_k) = x \circ (h_{l+1} \circ \cdots \circ h_k)$, thus only the sub-path from x to ω, which forms a minimal decomposition of $\omega \ominus x = x^{-1} \circ \omega$, needs to be found. Exploiting also the property that $|x| + |\omega \ominus x| = D$, $a \odot x$ can be computed by: (i) taking a minimal decomposition

s of $\omega \ominus x$, (ii) truncating s after $\lceil D - |\omega \ominus x| \rceil$ generators obtaining the sequence s', and (iii) computing $x \circ s'$.

Their implementations mainly require a randomized decomposition algorithm for the chosen generating set.

5. Randomized Decomposition Algorithms

Here we provide the last building blocks required to implement the differential mutation equation in order to work with permutations: the randomized decomposition algorithms for the three generating sets ASW, EXC, and INS.

Note that decomposing a permutation $x \in \mathcal{S}_n$ geometrically corresponds to finding the generators in a shortest path from e to x in the Cayley graph induced by the chosen generating set. However, since the items in e are in ascending order, it is more convenient to proceed in the opposite direction and then apply a simple algebraic transformation. Hence, the general scheme to obtain a decomposition of x is:

1. sort the items in x by using a minimal sequence s of adjacent swaps (in the ASW case), or exchanges (in the EXC case), or insertions (in the INS case);
2. reverse s and invert each generator, i.e., let $s = \langle h_1, h_2, \ldots, h_{l-1}, h_l \rangle$, then $\langle h_l^{-1}, h_{l-1}^{-1}, \ldots, h_2^{-1}, h_1^{-1} \rangle$ is a minimal decomposition of x.

Since we need a random minimal decomposition, the sorting stage at point 1 has to iteratively choose a random generator (that is, an elementary move) among those that bring the incumbent permutation closer to e.

In the rest of this section we describe, for each generating set, the corresponding randomized decomposition algorithm and the mechanisms required to multiply by a scalar greater than 1. Moreover, the computational complexity of the provided algorithmic procedures is analyzed in the worst-case scenario. Anyway, by considering that the expected weight of a randomly chosen permutation differs by a constant with respect to the weight of the maximal weight permutation, the same conclusions also apply to the average-case scenario.

5.1. Decomposition based on ASW generators

In order to introduce the ASW randomized decomposition algorithm, namely *RandBS*, we require the concept of *inversion*.

Given $x \in \mathcal{S}_n$, an ordered pair (i, j) is an inversion of x if and only if $i < j$ and $x(i) > x(j)$. The identity e is the unique permutation without inversions, thus the number of inversions, in a generic $x \in \mathcal{S}_n$, is a measure of the "unsortedness" of x. Moreover, if x has a non-empty set of inversions, then x has at least an inversion of the form $(i, i+1)$, i.e., an *adjacent inversion*. It is easy to show that, considering the generating set ASW: (1) $|x|$ equals the number of inversions of x, and (2) the inversion $(i, i+1)$ can be removed from x by composing it with the generator $\sigma_i \in ASW$.

```
1:  function RandBS(x ∈ S_n)
2:      s ← ⟨⟩                                   ▷ s is an empty sequence of generators
3:      A ← {σ_i ∈ ASW : x(i) > x(i + 1)}
4:      while A ≠ ∅ do                           ▷ A ≠ ∅ if and only if x ≠ e
5:          σ ← select a generator from A uniformly at random
6:          x ← x ∘ σ
7:          s ← Concatenate(s, ⟨σ⟩)
8:          A ← Update(A, σ)                     ▷ This step has O(1) complexity
9:      s ← Reverse(s)                           ▷ s is now a minimal decomposition of x
10:     return s
```

Figure 1: Randomized decomposition algorithms for ASW

Since the classical bubble-sort algorithm is known to sort a permutation using the minimal number of adjacent swaps, *RandBS*, previously introduced in [**? ?**], is implemented as a randomized variant of bubble-sort.

Its pseudo-code is provided in Figure 1. The set A (line 3) maintains the generators corresponding to the adjacent inversions of the incumbent permutation x. Then, x is iteratively sorted by applying a random generator from A (lines 5 and 6) which is appended to s (line 7). A is efficiently updated (line 8) by considering that σ_i removes the adjacent inversion $(i, i+1)$ and can only influence the two other adjacent inversions $(i-1, i)$ and $(i+1, i+2)$. Lastly, by considering that $\sigma^{-1} = \sigma$ for all $\sigma \in ASW$, reversing s (line 9) produces a minimal decomposition of the permutation in input. Since the diameter induced by ASW is $\binom{n}{2}$, *RandBS* has a worst-case complexity of $\Theta(n^2)$.

5.2. Maximal-weight permutation for ASW

In order to implement the multiplication for a scalar $a > 1$, the general scheme presented in Section 4.2 can be applied by considering that ASW has a unique maximal-weight permutation ω such that $x \sqsubseteq \omega$ for all $x \in \mathcal{S}_n$. The permutation ω is the reversed identity, i.e., $\omega = \langle n, n-1, \ldots, 1 \rangle$.

5.3. Decomposition based on EXC generators

In order to introduce the EXC randomized decomposition algorithm, namely *RandSS*, we require the concept of *cycle*.

A k-cycle of $x \in \mathcal{S}_n$ is a sequence of k items $(i_0, \ldots, i_{k-1})$ such that, for any $0 \leqslant j < k$, $x\left(i_{(j-1) \bmod k}\right) = i_j$. Any permutation can be uniquely represented by its cycle structure, e.g., $\langle 26745831 \rangle = (1268)(37)(4)(5)$. The identity e is the unique permutation with n cycles (of length 1), thus the number of cycles measures the "sortedness" of a permutation. Moreover, considering the generating set EXC, $|x|$ equals $n - \#\text{cycles}(x)$, for any $x \in \mathcal{S}_n$.

Note that if two items i_a and i_b belonging to the same cycle $(i_1, \ldots, i_a, \ldots, i_b, \ldots, i_k)$ are exchanged, the cycle is split into the two smaller cycles $(i_1, \ldots i_{a-1}, i_b, \ldots, i_k)$ and $(i_a, \ldots, i_{b-1})$. Therefore, *RandSS* iteratively sorts a permutation by applying a suitable exchange at every iteration.

Its pseudo-code is provided in Figure 2. The cycle structure of x at line 3 is computed in time $\Theta(n)$. The cycle weights w_i have been introduced in order to uniformly sample ϵ_{ij} among all the suitable exchanges (lines 7,8). Indeed,

```
1: function RandSS(x ∈ S(n))
2:     s ← ⟨⟩                                   ▷ s is an empty sequence of generators
3:     c ← getCycles(x)                         ▷ c_i is the ith cycle of x; c_ij is the jth item of cycle c_i
4:     for i ← 1 to len(c) do
5:         w_i ← len(c_i)(len(c_i) − 1)/2                                  ▷ weight of cycle c_i
6:     while len(c) < n do
7:         c_r ← randomly choose a cycle through a roulette wheel basing on the weights w_i
8:         i, j ← uniformly choose a pair of indexes from the cycle c_r
9:         x ← x ∘ ε_ij
10:        s ← Concatenate(s, ⟨ε_ij⟩)
11:        c ← Update(c, w)
12:    s ← ReverseInvert(s)                     ▷ s is now a minimal decomposition of x
13:    return s
```

Figure 2: Randomized decomposition algorithm for *EXC*

any k-cycle can be broken by using $\binom{k}{2}$ different exchanges (line 5). After the generator ϵ_{ij} is chosen, it is applied to x and appended to s (lines 9,10). Then, in line 11, the cycle structure is efficiently updated by replacing c_r with the longer between the two new sub-cycles, and appending the shorter at the end of the list. The loop at lines 6–11 performs no more than $n-1$ iterations and, by amortized analysis, the whole loop has a worst-case complexity of $\Theta(n)$. Since also the "reverse and invert" step of line 12 has linear complexity, *RandSS* costs $\Theta(n)$ in the worst case.

It is also interesting to note that *RandSS* generalizes the classical selection-sort algorithm. Indeed, it can be shown that selection-sort (when applied to permutations) works similarly to *RandSS* but with some limitations: it always breaks the cycle containing the smallest out-of-place item, and it always divides that k-cycle into two cycles of lengths 1 and $k-1$.

5.4. Maximal-weight permutations for EXC

In order to implement the multiplication by a scalar $a > 1$, note that Ω_{EXC} is formed by all the cyclic permutations, i.e., the permutations with only one cycle. However, given $x \in \mathcal{S}_n$, we need a procedure to compute an $\omega \in \Omega_{EXC}$ such that $x \sqsubseteq \omega$. With this aim, it is possible to proceed by iteratively merging two cycles of x until reaching a suitable cyclic permutation with the desired property. It turns out that two different cycles c_1, c_2 of a permutation can be merged by exchanging an item from c_1 with an item from c_2. Therefore, a procedure very similar to *RandSS*, and called *MergeCycles*, is devised.

Its pseudo-code is provided in Figure 3. In order to save computational time with respect to the general scheme described in Section 4.2, *MergeCycles* takes in input also the scalar $a > 1$ and directly computes the multiplication $a \odot x$. Indeed, since $a \odot x$ is in the path from x to the ω that we are computing, we simply need to stop the main loop when the number of cycles of the incumbent permutation is $\lceil a \cdot |x| \rceil$ (see line 6). Furthermore, since the procedure in line 11 can be efficiently computed without considering the order of the items inside the cycles, the cost of *MergeCycles* is $\Theta(n)$ in the worst case.

```
1: function MergeCycles(a > 1,x ∈ S(n))
2:     c ← getCycles(x)                          ▷ c_i is the ith cycle of x; c_ij is the jth item of cycle c_i
3:     for i ← 1 to len(c) do
4:         w_i ← len(c_i)                                                    ▷ weight of cycle c_i
5:     z ← x
6:     while |z| < ⌈a · |x|⌉ do                  ▷ |x| is equivalent to the number of cycles of x
7:         c_i, c_j ← randomly choose two different cycles basing on their weights
8:         i ← randomly choose an item from c_i
9:         j ← randomly choose an item from c_j
10:        z ← z ∘ ε_ij
11:        Merge(c_i, c_j)
12:    return z                                                              ▷ z = a ⊙ x
```

Figure 3: Multiplication by a scalar $a > 1$ for the *EXC* generating set

5.5. Decomposition based on INS generators

In order to introduce the *INS* randomized decomposition algorithm, namely *RandIS*, we require the concept of *longest increasing subsequence* (LIS).

Given $x \in \mathcal{S}_n$, an increasing subsequence of x is a sequence of items $(i_1, i_2, \ldots, i_k)$ such that $i_j < i_{j+1}$ and $x^{-1}(i_j) < x^{-1}(i_{j+1})$, for all $1 \leqslant j < k$. Then, a LIS is an increasing subsequence of x of maximal length. A LIS of x is not unique in general, however, the identity e is the only permutation with a single LIS of maximal length n, thus the LIS-length measures the "sortedness" of a permutation. Moreover, considering the generating set *INS*, $|x| = n - \text{LIS-length}(x)$ for any $x \in \mathcal{S}_n$.

Note that any LIS of a given permutation x can be extended by applying to x an insertion corresponding to a generator $\iota_{ij} \in \mathit{INS}$, such that: $x(i)$ is an item not appearing in the LIS, and j is taken from $[j_{min}, j_{max}]$ with j_{min} (j_{max}) indicating the position of the greatest (smallest) item of the LIS that is smaller (greater) than $x(i)$. For example, let $x = \langle 26745831 \rangle$ and choose the LIS (2458), then $x \circ \iota_{72}$ shifts the item $x(7) = 3$ to position 2 in x. Thus, the new permutation is $x \circ \iota_{72} = \langle 23674581 \rangle$ and its LIS is (23458). Hence, the LIS has been effectively extended with one more item.

RandIS, presented in Figure 4, randomly computes a LIS of the input permutation, and iteratively applies a random insertion that extends the LIS.

The random LIS L is computed at line 3 by using a simple stochastic variant of the algorithm provided in [**?** , Sec. 2]. Its complexity is $O(n \log n)$. The set U (line 4) is formed by the items not appearing in L and that need to be iteratively shifted "into" L. In order to uniformly sample ι_{ij} among all the suitable insertions (lines 9–11), any item in U is weighted by the number of suitable insertions in which it is involved (lines 5–7). At every iteration of the main loop, the generator ι_{ij} is chosen, applied to x and appended to s (lines 9–13). The loop stops when $\text{len}(L) = n$, $U = \varnothing$ and $x = e$, therefore no more than $n - 1$ iterations are performed. The most costly operations of the loop are the updates at lines 14 and 15. However, since the sets $P^L_{x,k}$ are actually intervals, only their end-points need to be maintained, hence the updates, as also the last "reverse and invert" step of line 16, can be done in linear time. Finally, since no more than $n-1$ iterations are required, the overall complexity of *RandIS* is $\Theta(n^2)$ in the worst case.

```
1: function RandIS(x ∈ S(n))
2:     s ← ⟨ ⟩                                   ▷ s is an empty sequence of generators
3:     L ← getRandomLIS(x)
4:     U ← {1, ..., n}\L                          ▷ Set of items not appearing in L
5:     for all k ∈ U do
6:         P^L_{x,k} ← set of positions in x where it is possible to shift item k in order to extend L
7:         w_k ← |P^L_{π,k}|                      ▷ weight of item k
8:     while len(L) < n do
9:         r ← randomly choose an item in U through a roulette wheel basing on the weights w_k
10:        i ← x^{-1}(r)                          ▷ The position of r in x
11:        j ← randomly choose a position from P^L_{x,r}
12:        x ← x ∘ ι_{ij}
13:        s ← Concatenate(s, ⟨ι_{ij}⟩)
14:        Update(L, U)
15:        Update(P^L_{x,k}, w_k) for any k ∈ U
16:    s ← ReverseInvert(s)                       ▷ s is now a minimal decomposition of x
17:    return s
```

Figure 4: Randomized decomposition algorithm for *INS*

It is also interesting to note that *RandIS* generalizes the classical insertion-sort algorithm. Indeed, insertion-sort iteratively extends the increasing subsequence maintained at consecutive indexes in the leftmost part of the permutation, while *RandIS* allows to spread the increasing subsequence anywhere in the permutation.

5.6. INS approximate multiplication by $a > 1$

In order to implement the multiplication by a scalar $a > 1$ for the *INS* generating set, we could be tempted to consider the unique maximal-weight permutation $\omega = \langle n, n-1, \ldots, 1 \rangle$ and use the general scheme described in Section 4.2. It is easy to see that ω has a minimal LIS-length of 1 and that, for any $x \in \mathcal{S}_n$, $|x| = n - \text{LIS-length}(x)$. Hence, when $a > 1$, since $a \odot x$ has to move x closer to ω, we need to find insertions that reduce the LIS-length of x but, unfortunately, there exists permutations where this is impossible. For instance, it is easy to show that none of the 9 insertions of $\mathcal{S}_4$ reduce the LIS-length of the permutation $\langle 2413 \rangle$. This issue is a consequence of the non-uniqueness of the LISs. In other words, it means that not all the permutations $x \in \mathcal{S}_n$ satisfy the requirement $x \sqsubseteq \omega$, thus conditions (C1) and (C3) cannot be satisfied at the same time.

However, we can consider the concept of *longest decreasing subsequence* (LDS). The LDS is somehow antipodal to the LIS, i.e., given $x \in \mathcal{S}_n$, a decreasing subsequence of x is a sequence of items $(i_1, i_2, \ldots, i_k)$ such that $i_j > i_{j+1}$ and $x^{-1}(i_j) < x^{-1}(i_{j+1})$, for all $1 \leqslant j < k$. Analogously to the LIS case, there always exists an insertion that increases the LDS-length of a generic permutation. Anyway, in this case the permutation gets closer to ω.

Using this observation, it is possible to directly employ *RandIS* to implement the approximate multiplication by $a > 1$. The weight $|x| = n - \text{LIS-length}(x)$ in condition (C1) is replaced with the surrogate weight $|x|^* = \text{LDS-length}(x) - 1$. Note that $|\omega|^* = |\omega| = n - 1$, $|e|^* = |e| = 0$, and, more generally, $|x|^*$ is a non-strict monotone transformation of $|x|$ that, conversely from the latter,

can be increased by a suitable insertion. Therefore, we can iteratively apply an insertion that increases $|x|^*$. Finally, let s^R be the reverse of a sequence s, for all $x \in \mathcal{S}_n$, a LDS of x can be obtained by reversing a LIS of x^R, i.e., $\mathrm{LDS}(x) = (\mathrm{LIS}(x^R))^R$. Therefore, an insertion ι_{ij} increases the LDS-length of x if and only if the insertion $\iota_{n+1-i,n+1-j}$ increases the LIS-length of x^R.

6. Variable Neighborhood Algebraic Differential Evolution for Permutations

Based on the previously presented algebraic tools, here we introduce the Variable Neighborhood Algebraic Differential Evolution for Permutations (VNDEP). Its main scheme is provided in Figure 6 and follows the general scheme of classical DE (see Section 2.1).

```
1:  function VNDEP
2:      Initialize Population x_1, ..., x_N
3:      while evaluations budget is not exhausted do
4:          for i ← 1 to N do
5:              Choose the scale factor F and the crossover strength CR
6:              Choose the generating set H and the crossover operator C
7:              y_i ← DifferentialMutation(i, F, H)
8:              z_i ← C(x_i, y_i, CR)
9:              Evaluate f(z_i)
10:         for i ← 1 to N do
11:             x_i ← Selection(x_i, z_i)
12:         if restart criterion is verified then
13:             Restart the Population
14:     return the best permutation found
```

Figure 5: Differential Evolution for Permutations

VNDEP aims to minimize a given objective function $f : \mathcal{S}_n \to \mathbb{R}$ by iteratively evolving a population of N permutations. The evolution is performed by means of the three genetic operators: differential mutation, crossover, and selection. The DE key operator, i.e., the differential mutation, is designed by exploiting the algebraic concepts previously presented. Two variants of widely used crossover schemes for permutations are considered, while the one-to-one selection works as in classical DE.

VNDEP is endowed with adaptive mechanisms that self-regulate its parameters: the scale factor F, the crossover strength CR, the generating set H, and the crossover scheme C. Therefore, the population size N needs to be set. Finally, the diversity loss phenomenon, typical of combinatorial problems [**?**], is mitigated by introducing a restart procedure.

All the VNDEP components are described in the following sections.

6.1. Algebraic Differential Mutation

The algebraic differential mutation operator is designed by exploiting the algebraic concepts previously presented.

Hence, it is possible to rewrite any differential mutation strategy of the classical DE by using the vector-like operations introduced in Sections 4 and 5.

For example, *rand/1* (see Equation (3)) for the i–th population individual can be rewritten to directly work with permutations as

$$y_i = x_{r_1} \oplus F \odot (x_{r_2} \ominus x_{r_3}) \tag{8}$$

where, as in classical DE, the scale factor $F > 0$ regulates the strength of the mutation, while r_1, r_2, r_3 are three indexes in $\{1, \ldots, N\}$ different from each other and from i. For the properties described in Sections 3 and 4, this discrete operation simulates, in the Cayley graph of permutations, the geometric properties of its continuous counterpart.

By taking inspiration from the external archive introduced in [**?**], VNDEP uses a new variant of *rand/1* where x_{r3} is randomly sampled from the set $(\{x_1, \ldots, x_N\} \backslash \{x_i, x_{r_1}, x_{r_2}\}) \cup \mathcal{A}$. $\mathcal{A}$ is the external archive that contains the most recently discarded (by the selection) population individuals. $\mathcal{A}$ has maximum size N as the main population and, when it is full, the last discarded solution from the main population replaces a random element of $\mathcal{A}$. The aim of the external archive is to increase the diversity of the mutants produced by the differential mutation scheme.

Furthermore, differently from the continuous case, for any mutation strategy there are three possible implementations that depend on the chosen generating set $H \in \mathcal{H}$. In principle, any single application of the differential mutation can use a different generating set. Hence, the adaptive scheme described in Section 6.3 is used to automatically select H during the evolution.

6.2. Crossovers for Permutations

We consider two popular crossovers for the permutation representation, namely, the position based crossover *POS* [**?**] and the two point crossover *TPII* [**?**].

Let $x, y \in \mathcal{S}_n$ denote, respectively, the parent and mutant permutations that have to be recombined, then both *POS* and *TPII* select a random subset of positions $P \subseteq [n]$ and build an offspring $z \in \mathcal{S}_n$ by: (1) setting $z(i) \leftarrow y(i)$ for any $i \in P$, and (2) inserting the remaining items starting from the leftmost free place of z and following the order of appearance in x.

The difference between *POS* and *TPII* is in the way the set P is built: P is an interval of consecutive positions in *TPII*, while it can be any subset in *POS* .

Furthermore, both *POS* and *TPII* have been modified in order to introduce the DE parameter $CR \in [0, 1]$. The modified variants, namely POS^{CR} and $TPII^{CR}$, constrain the size of P to $|P| = \lceil CR \cdot n \rceil$.

It is interesting to note that POS^{CR} and $TPII^{CR}$ can be seen as feasible variants, for the permutation representation, of, respectively, the binomial and exponential crossovers of classical DE.

We denote by $\mathcal{C}$ the set of crossover operators $\left\{POS^{CR}, TPII^{CR}\right\}$, and, in Section 6.3, we describe an adaptive scheme that allows to automatically select a crossover $C \in \mathcal{C}$ during the evolution.

6.3. Adaptive schemes

By recalling that $\mathcal{H} = \{ASW, EXC, INS\}$ and $\mathcal{C} = \{POS^{CR}, TPII^{CR}\}$, VNDEP self-regulates four parameters: $F > 0$, $CR \in [0, 1]$, $H \in \mathcal{H}$, and $C \in \mathcal{C}$.

For F and CR it is possible to use any one of the numerical DE adaptive schemes already proposed in literature (see Section 2.1). However, due to the different characteristics of continuous and combinatorial spaces, it is not guaranteed that good adaptive schemes for numerical problems are also good for permutation problems. Hence, we have conducted some preliminary experiments in order to compare SHADE and jDE. The results clearly indicate that jDE [**?**] has to be preferred in our case. Moreover, we also observed that a value $F > 1$ is beneficial to the search, in particular when the population starts to stagnate. Therefore, we have used the jDE scheme described in Section 2.1 with a slight modification: F is sampled from the extended interval $[0.1, 1.25]$, thus a difference in the differential mutation can be, not only scaled down, but also extended till the 125% of its length.

Conversely, H and C are separately adapted by means of a novel scheme based on dynamic rewards to which we refer as *REW-adaptation*. In the following we describe the application of REW-adaptation to H. The case of C is analogous.

REW-adaptation returns a different generating set H (and a crossover operator C) for any individual according to a probabilistic choice.

For any $H \in \mathcal{H}$, two values are maintained during the evolution: the accumulated reward ar_H, and the number m_H of times that H has been chosen. Hence, the expected reward for H is defined as $\overline{r}_H = ar_H/m_H$. Then, a probability distribution on $\mathcal{H}$ is generated by a mixture between a softmax function [**?**] and a uniform distribution. Formally, the probability of choosing H is given by

$$p_H = (1 - \eta)\frac{e^{\overline{r}_H}}{\sum_{H' \in \mathcal{H}} e^{\overline{r}_{H'}}} + \eta\frac{1}{|\mathcal{H}|}. \tag{9}$$

The softmax part exponentially amplifies the differences in terms of expected rewards among the generating sets, while the uniform part assigns a small chance to be selected to any generating set, thus avoiding the irreversible situation of a single generating set with all the probability mass. The parameter $\eta \in [0, 1]$ regulates the importance of the uniform distribution in the mixture. We empirically chose to set $\eta = 0.1$.

Furthermore, at the end of every iteration, m_H is incremented for every use of H, while a reward is assigned for any successful application of a generating set. Indeed, when a trial individual, produced by means of H, passes the selection, a reward r_{it} is assigned to H, i.e., $ar_H \leftarrow ar_H + r_{it}$.

The reward is dynamically assigned as $r_{it} = it \cdot s$ according to the iteration number it and to a scaling factor s that has been experimentally set to the constant value $s = 0.1$. It is important to note that this scheme assigns a greater reward to later improvements. This choice is motivated by the behavior observed in all the evolutionary algorithms, where it is easy to produce an

improvement in earlier iterations, while it gets progressively more difficult as the evolution goes on.

6.4. Restart procedure

Since the permutations space is finite, the chances that the population constantly looses diversity and finally converges to a super-individual, i.e., a population composed by N copies of the same permutation, are not negligible. Ignoring the external archive in the following analysis, it is easy to see that, in VNDEP, when the population has converged to a super-individual, the offsprings produced by mutation and crossover are again equal to the super-individual. Hence, the evolution gets stuck.

The external archive introduced in Section 6.1 mitigates in part this problem, but we empirically observed that a more drastic solution is sometimes required. Therefore, we introduce a restart mechanism that is executed when all the individuals have the same fitness (which is roughly equivalent to check for a super-individual, but with a smaller computational cost). Then, the restart procedure replaces, with randomly generated permutations, all the individuals except one. Furthermore, the external archive $\mathcal{A}$ is emptied and the REW-adaptation variables ar_H and m_H are reset.

7. Experiments

Experiments have been held on commonly adopted benchmark suites for LOPCC: UMTS (100 instances with $n = 16$), LOLIB (44 instances with $44 \leqslant n \leqslant 60$), and RND (three sets of 25 instances of size 35, 100, and 150). All the instances are available at *www.optsicom.es/lopcc*.

In order to verify the effectiveness of our proposal with respect to classical DE schemes for permutation problems, VNDEP has been compared with the two DE variants jDE and SHADE endowed with both the permutation-based decoder functions RK and RKI. Therefore, we have experimentally compared the following five algorithms: VNDEP, jDE-RK, jDE-RKI, SHADE-RK, and SHADE-RKI. The results of this comparison are provided and discussed in Section 7.1.

The best results obtained by VNDEP are further analyzed in Section 7.2 where a comparison with state-of-the-art results is provided. It is important to note that, in this way, VNDEP is implicitly compared with respect to all the algorithms which have successfully approached the problem [**? ? ? ? ?**].

Finally, additional experiments are described in Section 7.3. Here, the aim is twofold: analyzing the performance differences of VNDEP with respect to the previous non-adaptive DEP implementations, and studying the effectiveness of VNDEP on the well known instances of the Linear Ordering Problem (LOP) without cumulative costs.

7.1. Comparison with random-key DEs

The aim of this section is to compare VNDEP with respect to the four random-key based DE schemes jDE-RK, jDE-RKI, SHADE-RK, and SHADE-RKI (see Section 2).

In all the five algorithms, only the population size N requires to be set. In order to perform a fair comparison, N has been separately tuned for each algorithm. A variety of tools and techniques for parameter tuning can be adopted like, for instance, iRACE [**?**] and SMAC [**?**]. In this work, we have used SMAC [**?**]. The considered values for N are $20, 30, 50, 80$ and 100. The SMAC calibrations have been run using the first five instances (in lexicographical order) in any benchmark suite, thus 30 instances in total. Every SMAC calibration has been set to perform 1000 executions, while every execution terminates after $50\,000n$ fitness evaluations have been performed. Quite surprisingly, all the five calibrations suggest the same population size, i.e., $N = 80$.

After the tuning phase, each competitor has been run 20 times per instance, thus a total of 31 400 executions have been performed. As in the calibration phase, every execution terminates after $50\,000n$ evaluations. The final fitness values produced by the executions of every algorithm have been aggregated for each instance by using the Average Relative Percentage Deviation (ARPD) measure which is computed according to

$$ARPD^{Alg}_{Inst} = \frac{1}{20}\sum_{i=1}^{20}\frac{\left|Alg^{(i)}_{Inst} - Best_{Inst}\right|}{Best_{Inst}} \times 100 \tag{10}$$

where $Alg^{(i)}_{Inst}$ is the final fitness value produced by the algorithm Alg in its i-th run on the instance $Inst$, while $Best_{Inst}$ is the best result obtained by any algorithm in any run on the given instance.

Table 1 provides the ARPDs and the average ranks aggregated on every benchmark suite (computed *à la Friedman* as explained, for example, in [**?**]). Note that, for the sake of presentation, the RND suite have been further divided with respect to the instance size. The best results are highlighted in bold.

Table 1: Experimental comparison on LOPCC benchmarks

Benchmarks	**VNDEP**		**jDE-RK**		**jDE-RKI**		**SHADE-RK**		**SHADE-RKI**	
	ARPD	Rank	ARPD	Rank	ARPD	Rank	ARPD	Rank	ARPD	Rank
UMTS	**0.00**	**1.34**	0.55	3.17	0.07	2.02	1.63	4.31	0.72	4.17
RND35	**0.00**	**1.00**	23.10	4.04	4.29	2.00	29.01	4.96	13.78	3.00
LOLIB	**0.00**	**1.00**	258.17	3.98	11.67	2.05	808.02	4.77	65.30	3.21
RND100	**1.29**	**1.00**	1251.21	4.00	120.82	2.04	2009.14	5.00	202.44	2.96
RND150	**13.75**	**1.00**	36662.86	4.00	616.74	2.52	99962.96	5.00	662.91	2.48
Overall	**1.91**	**1.15**	4382.85	3.62	56.83	2.08	11807.05	4.64	73.81	3.51

The table clearly shows that VNDEP largely outperforms all the random-key schemes. Indeed, VNDEP obtains an average rank of 1 in almost all benchmark suites. The only exception is on the smallest instances UMTS ($n = 16$), where also the other competitors have been able to reach some of the (presumably) optima. In particular, on the three smaller LOPCC benchmark suites, VNDEP

produced 0 as ARPD, thus meaning that all the VNDEP executions obtained the best result in every instance. On the more difficult RND100 and RND150 instance sets, VNDEP is slightly less robust than in the other cases (ARPDs larger than 0), however its difference with respect to the random-key DEs is huge.

Finally, as secondary aspects: the jDE scheme looks better than SHADE on permutation problems, while, regarding the random-key variants, RKI outperforms RK.

7.2. Comparison with state-of-the-art results

In order to validate VNDEP performances with respect to non-DE schemes, here we compare its best results obtained on every tested instance with the state-of-the-art results available in literature.

For LOPCC, the true optima of the instances are not known, except for the smaller UMTS benchmark. Hence, the best known solutions have been collected by aggregating the results of the state-of-the-art algorithms presented in [**?**], [**?**], [**?**], [**?**], and the more recent [**?**].

VNDEP reached the best known solution in every run on all the small instances, i.e., UMTS, RND35, and LOLIB. More interesting are the comparisons on the larger instance sets RND100 and RND150 that are provided in Tables, respectively, 2 and 3. For every instance, the (previous) best known and the best VNDEP fitness value together with the relative percentage deviation between them are provided. VNDEP fitness values are in bold or italic when it, respectively, improves or matches, the best known fitness.

The most impressive datum shown by Tables 2 and 3 is that, of a total of 50 instances, VNDEP obtained 32 new state-of-the-art solutions, while it has been able to at least match the known optima, in 40 instances. In average, VNDEP improved by 0.32% the previously best known results of RND100 instances and by 1.90% those of RND150 instances.

7.3. Additional experiments

We analyze the performances of VNDEP with respect to the 10 non-adaptive DEP implementations experimented in [**?**]. In order to make a more concise presentation, we compare, on every LOPCC instance, the best objective value obtained by VNDEP with respect to the best objective value obtained in all the executions of all the 10 non-adaptive DEP implementations. The comparison is summarized in Table 4 where, for every benchmark set, it is provided: the number of instances where VNDEP outperformed DEP ($\blacktriangle$), the number of instances where VNDEP and DEP obtained the same result ($=$), the number of instances where VNDEP is outperformed by DEP ($\blacktriangledown$), and the average improvement of VNDEP with respect to DEP.

These data clearly show that VNDEP is never worse than the best DEP setting. Most notable are the results for the RND100 and RND150 benchmarks, where all the 50 objective values obtained by DEP have been strictly improved by VNDEP. Moreover, though not shown in Table 4, the average objective

Table 2: Comparison with best known results on LOPCC RND100 instances

Instance	**Best Known**	**VNDEP**	**Rel. Dev.**
t1d100.1	246.279	*246.279*	0.00
t1d100.2	284.924	**282.933**	−0.70
t1d100.3	1236.237	**1230.211**	−0.49
t1d100.4	6735.661	*6735.661*	0.00
t1d100.5	162.261	**159.808**	−1.51
t1d100.6	391.662	**390.943**	−0.18
t1d100.7	5641.137	*5641.137*	0.00
t1d100.8	2750.802	*2750.802*	0.00
t1d100.9	61.772	**61.425**	−0.56
t1d100.10	155.892	**154.812**	−0.69
t1d100.11	227.877	230.346	+1.08
t1d100.12	231.176	*231.176*	0.00
t1d100.13	577.453	591.506	+2.43
t1d100.14	246.030	**242.554**	−1.41
t1d100.15	406.478	*406.478*	0.00
t1d100.16	707.413	*707.413*	0.00
t1d100.17	715.613	**713.576**	−0.28
t1d100.18	621.415	**620.809**	−0.10
t1d100.19	227.374	*227.374*	0.00
t1d100.20	236.088	**231.865**	−1.79
t1d100.21	221.462	**220.144**	−0.60
t1d100.22	141.255	**140.005**	−0.88
t1d100.23	1588.314	**1581.866**	−0.41
t1d100.24	464.961	**461.479**	−0.75
t1d100.25	632.586	**625.483**	−1.89
Average Relative Deviation			**−0.32**

Table 3: Comparison with best known results on LOPCC RND150 instances

Instance	**Best Known**	**VNDEP**	**Rel. Dev.**
t1d150.1	8293.108	8698.638	+4.89
t1d150.2	159339.130	**144514.109**	−9.30
t1d150.3	548507.282	574549.081	+4.75
t1d150.4	68125.331	69808.729	+2.47
t1d150.5	75426.662	**67009.770**	−11.16
t1d150.6	44961.697	**42502.756**	−5.47
t1d150.7	150146.763	**149093.003**	−0.70
t1d150.8	247564.438	248881.062	+0.53
t1d150.9	363221.346	**352938.663**	−2.83
t1d150.10	107685.011	**100831.369**	−6.36
t1d150.11	12360.337	**11915.362**	−3.60
t1d150.12	60614.534	62222.772	+2.65
t1d150.13	91988.932	**91267.029**	−0.78
t1d150.14	70153.934	72484.726	+3.32
t1d150.15	321468.489	**297746.166**	−7.38
t1d150.16	16231674.691	**15364771.070**	−5.34
t1d150.17	71190.802	**68346.418**	−4.00
t1d150.18	629986.069	**610051.969**	−3.16
t1d150.19	59594.204	**59554.467**	−0.07
t1d150.20	1886041.875	1993411.258	+5.69
t1d150.21	39248.997	**38327.650**	−2.35
t1d150.22	671281.287	**651354.266**	−2.97
t1d150.23	21468279.568	**18983183.660**	−11.58
t1d150.24	100543.430	**96832.862**	−3.69
t1d150.25	462316.511	504169.325	+9.05
Average Relative Deviation			**−1.90**

Table 4: Comparison between VNDEP and non-adaptive DEP

Benchmarks	▲	=	▼	**Avg Impr.**
UMTS	0	100	0	0%
RND35	3	22	0	0.2532%
LOLIB	5	39	0	0.0005%
RND100	25	0	0	2.0443%
RND150	25	0	0	7.3271%
Overall	58	161	0	1.0728%

value obtained by VNDEP executions is always better than those obtained by the best non-adaptive DEP scheme. Therefore, the adaptive scheme used in VNDEP represents a consistent improvement with respect to the non-adaptive approaches previously proposed in [**?**].

We further study the contribution of each generating set in a typical execution of VNDEP. In this regard, Figure 6 shows the behavior of the probability assigned to every generating set by the *REW-adaptation* scheme in an execution of VNDEP on the instance `t1d100.1`. We have noted that all the observed executions of VNDEP show a similar behavior.

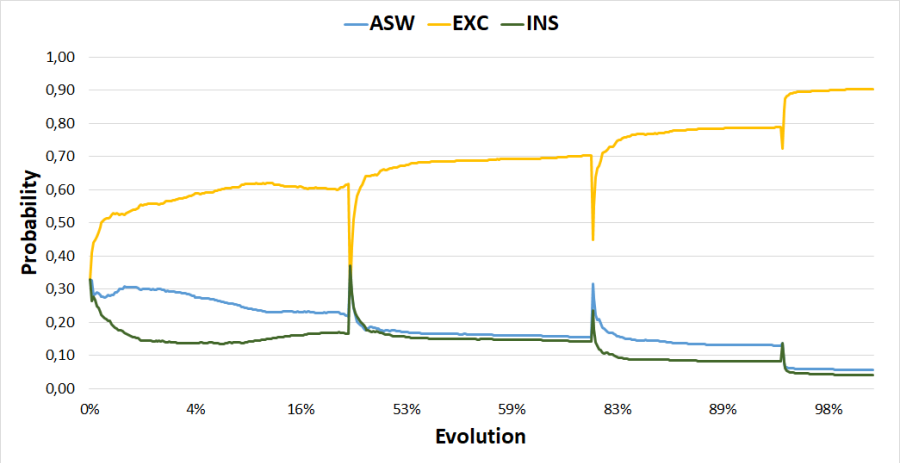

Figure 6: Probabilities assigned to generating sets during the evolution.

Since *REW-adaptation* assigns probabilities based on the successes obtained during the evolution, Figure 6 shows that the *EXC* generating set clearly gives a larger contribution than *ASW* and *INS*.

However, since VNDEP outperformed even the non-adaptive DEP implementation using *EXC*, also the contributions of *ASW* and *INS* are important in order to reach good objective values.

Finally, in order to show the generality of the proposed approach, we also executed VNDEP on instances of the standard version of the Linear Ordering Problem (LOP) [**?** **?**]. Three benchmark suites have been considered: IO (50

instances with $44 \leqslant n \leqslant 79$), SGB (25 instance with $n = 75$), and MB (30 instances with $100 \leqslant n \leqslant 250$). These instances have known optima (available at *www.optsicom.es/lolib*), thus making it possible to measure how good VNDEP is in an absolute sense. VNDEP has been executed 20 times per instance using the same setting described in Section 7.1. Table 5 synthesizes the experiments by showing, for each benchmark suite, the number of instances where the optimum has been reached by VNDEP in at least one execution and in all the 20 executions.

Table 5: VNDEP performances on LOP instances

Benchmarks	**#Instance solved in at lest one run**	**#Instance solved in all the runs**
IO	50/50	50/50
SGB	25/25	24/25
MB	30/30	27/30
Overall	105/105	101/105

Interestingly, VNDEP has been able to reach the optimum in every instance. Moreover, in 101 instances (out of 105) the optimum has been reached in all the 20 executions of VNDEP, while in the remaining four instances (`N-sgb75.02`, `N-r100c2`, `N-r150b1` and `N-r200e1`), the optimum has been reached in more than 14 executions (out of 20).

8. Conclusion and Future Work

We presented a Variable Neighborhood Algebraic Differential Evolution algorithm for permutation-based optimization problems.

VNDEP individuals simultaneously search in the three classical permutation neighborhoods by using a novel adaptive mechanism to dynamically choose a generating set during the evolution.

We have described and analyzed the decomposition algorithms for all the proposed generating sets. For the generating set based on insertion moves, an approximated method for the multiplication by a scalar $a > 1$ has been introduced. Moreover, the dynamical adaptation mechanism has been extended also to the crossover scheme and the scale factor parameter, thus VNDEP only requires to set the population size.

As a case of study, VNDEP has been applied to the linear ordering problem with cumulative costs. Experiments have been held on a wide set of commonly adopted benchmark instances, where VNDEP has been compared with the random-key based DEs, which are widely used in literature to tackle permutation problems. Furthermore, a comparison with the state-of-the-art results has been carried out.

VNDEP largely outperforms the four random-key competitor algorithms and, most remarkably, it obtained 32 new best known solutions of the 50 most challenging instances. Moreover, experiments have been held also on commonly

adopted instances of the LOP, where VNDEP reached, in almost every execution, the known optimal values.

As future work, we are planning to further analyze the adaptive mechanism based on the dynamic rewards and to use it in other evolutionary algorithms. Finally, we are also interested to apply VNDEP to other permutation-based optimization problems.

Acknowledgements

The research described in this work has been partially supported by: the research grant "Fondi per i progetti di ricerca scientifica di Ateneo 2019" of the University for Foreigners of Perugia under the project "Algoritmi evolutivi per problemi di ottimizzazione e modelli di apprendimento automatico con applicazioni al Natural Language Processing"; 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