Theoretical and Empirical Analysis of Lehmer Codes to Search Permutation Spaces with Evolutionary Algorithms

Yuxuan Ma[1], **Valentino Santucci**[2], **Carsten Witt**[3]
[1]Southern University of Science and Technology, China
[2]University for Foreigners of Perugia, Italy
[3]Technical University of Denmark, Denmark
12212608@mail.sustech.edu.cn,valentino.santucci@unistrapg.it,cawi@dtu.dk

Abstract

A suitable choice of the representation of candidate solutions is crucial for the efficiency of evolutionary algorithms and related metaheuristics. We focus on problems in permutation spaces, which are at the core of numerous practical applications of such algorithms, e. g. in scheduling and transportation. Inversion vectors (also called Lehmer codes) are an alternative representation of the permutation space S_n compared to the classical encoding as a vector of n unique entries. In particular, they do not require any constraint handling. Using rigorous mathematical runtime analyses, we compare the efficiency of inversion vector encodings to the classical representation and give theory-guided advice on their choice. Moreover, we link the effect of local changes in the inversion code space to classical measures on permutations like the number of inversions. Finally, through experimental studies on linear ordering and quadratic assignment problems, we demonstrate the practical efficiency of inversion vector encodings.

1 Introduction

Permutations are fundamental algebraic objects with broad applications, as they can model diverse concepts such as orderings or rankings of items, bijective mappings between two sets of items, and tours or sets of cycles within a collection of locations. Because of their versatility, permutations form the solution space for many combinatorial optimization problems. These problems, often called *permutation problems* (Santucci and Ceberio 2025), include notable examples such as: the Hamiltonian Path Problem (HPP), which involves finding the shortest path visiting every given location exactly once; the Linear Ordering Problem (LOP), which seeks for a maximum-weight directed acyclic subgraph within a given digraph; and the Quadratic Assignment Problem (QAP), which aims to determine a cost-minimizing assignment of facilities to locations. Permutation problems find applications in many tasks of practical relevance. Among others, the LOP has been applied to model consensus ranking in computational social choice (Marti and Reinelt 2022) and to improve natural language translations (Tromble and Eisner 2009), while the QAP has been used for graph matching tasks (Wang, Yan, and Yang 2021). Therefore, there is interest in developing effective and efficient solvers for permutation problems.

Nonetheless, permutation problems pose significant challenges. First, they are often NP-hard, as are all the aforementioned examples. Second, even among NP-hard problems, they are particularly challenging because the size of their search space increases factorially with the instance size—i.e., even faster than any exponential function. This inherent difficulty has motivated the adoption of non-traditional algorithmic approaches, including metaheuristics, evolutionary algorithms (EAs) and, more recently, also neural networks and deep learning methodologies.

However, the constrained nature of permutations poses representation challenges for these methods. The most common encoding scheme is the classical linear encoding, which represents a permutation as a vector of unique items satisfying mutual exclusivity. This constraint is typically handled in EAs and other metaheuristics through specifically designed neighborhood structures (Schiavinotto and Stützle 2007) and variation operators (Santucci, Baioletti, and Milani 2015). An alternative encoding is the permutation matrix, which represents a permutation as a binary matrix with exactly one 1-entry per row and column. This encoding is particularly popular in neural network architectures that generate permutations, where the final layer applies a temperature-decreasing Sinkhorn operator (Mena, Belanger, Linderman, and Snoek 2018) to iteratively transform the activation values of n^2 neurons into a permutation matrix.

To overcome the mutual exclusivity constraint, *random key* representations were introduced (Bean 1994). In this encoding scheme, a real-valued vector can be decoded to a permutation by either argsorting its elements or replacing its values with their ranks within the vector. However, a single permutation may correspond to an infinite number of vectors, leading to redundancy. Additional criticisms to this approach are discussed by Santucci, Baioletti, and Milani (2020). Another method to get rid of mutual exclusivity is to reformulate the permutation problem in the parameter space of a differentiable probability model over permutations (Ceberio and Santucci 2023). However, this approach typically requires more sophisticated algorithms.

Interestingly, there exists a less commonly used encoding scheme for permutations, called *Lehmer code* or *inversion vector* (Lehmer 1960), that: (i) does not require to satisfy any

constraints, aside from trivial domain bounds, (ii) is in one-to-one correspondence with permutations, and (iii) does not necessarily require sophisticated algorithms to work with. Given a permutation, its Lehmer code is a vector of integers, where each entry indicates how many smaller items appear after that position in the original linear permutation.

This work investigates the effectiveness of Lehmer codes in EAs, with a primary emphasis on theoretical analysis. Our main theoretical contributions are threefold: we define a series of benchmark problems in the Lehmer code space that are closely related to current work in runtime analysis; we introduce simple algorithms for searching within this space; and, most importantly, we derive runtime bounds for these algorithms on the proposed benchmarks, contrasting them with existing algorithms and benchmarks defined over the traditional permutation space. Along the way, we improve an existing runtime analysis by Doerr and Pohl (2012) for a related, multi-valued search space by a factor of almost $\Theta(n^2)$. In addition, we also study the structural connections between Lehmer space and permutation space. To the best of our knowledge, this is the first study to explore these directions. To complement this, we also conduct an empirical evaluation on real-world instances of two different types of NP-hard permutation problems: the LOP, an ordering problem, and the QAP, an assignment problem. The source code is available at https://github.com/TrendMYX/LehmerEA.

1.1 Related Work

Lehmer code representations in EAs have mostly been studied empirically so far (Regnier-Coudert and McCall 2014; Marmion and Regnier-Coudert 2015; Khalil and Abdou 2021; Uher and Kromer 2022), showing scenarios where they are preferable to classical encodings.

The first work describing theoretical runtime analyses of EAs on permutation problems appeared 20 years ago (Scharnow, Tinnefeld, and Wegener 2005). However, since then, the major focus of runtime analyses of EAs has been on binary search spaces (see Doerr and Neumann 2020 for an overview) and results for permutations were only sporadic, see, e. g., Gavenciak, Geissmann, and Lengler (2019); Bassin and Buzdalov (2020). Only recently, there has been increasing momentum in this area, by Doerr, Ghannane, and Ibn Brahim (2023) suggesting a new framework for the runtime analysis and Baumann, Rutter, and Sudholt (2024, 2025) analyzing permutation spaces arising in graph drawing problems. However, these studies all use the canonical representation of permutations as a vector of n unique entries and to the best of our knowledge, there has been no runtime analysis for other representations of permutations so far. Our study of Lehmer codes fills this gap.

2 Preliminaries

2.1 Concepts and Notation

Let $[a..b] := \{z \in \mathbb{Z} \mid a \leq z \leq b\}$ and $[k] := [0..k-1]$. A mapping from $[1..n]$ to itself is called a *permutation* (of $[1..n]$) if it is bijective. We denote by S_n the set of all permutations of $[1..n]$. A common notation represents a permutation $\sigma \in S_n$ as a vector $(\sigma(1), \sigma(2), \ldots, \sigma(n))$. The *identity* permutation in S_n is $(1, 2, \ldots, n)$. For clarity, we assume that the vector representation of a permutation is 1-indexed from the left.

As in Doerr, Ghannane, and Ibn Brahim (2023), we adopt the standard composition $\circ$ of permutations: for $\sigma, \tau \in S_n$, $\tau \circ \sigma$ is defined by $(\tau \circ \sigma)(i) = \tau(\sigma(i))$ for all $i \in [1..n]$.

Another common notation is the *cycle* notation, where a *cycle* of length k (a k-cycle) is a permutation $\sigma \in S_n$ such that there exist k distinct elements $i_1, \ldots, i_k \in [1..n]$ satisfying: for all $j \in [1..k-1]$, $\sigma(i_j) = i_{j+1}$, and $\sigma(i_k) = i_1$; while for all $j \in [1..n] \setminus \{i_1, \ldots, i_k\}$, $\sigma(j) = j$. We denote such a cycle by $\sigma = (i_1\ i_2\ \cdots\ i_k)$. Any permutation can be written as a product of disjoint cycles. A 2-cycle is called a *transposition*. A transposition of the form $s_i = (i\ i{+}1)$ is called an *adjacent transposition*.

Note that an adjacent transposition $s_i = (i\ i{+}1)$ acts as an *adjacent swap* of the values at indices i and $i+1$ of a permutation $\sigma \in S_n$ when σ is right-multiplied by s_i, i.e., $\sigma \circ s_i$. In Scharnow, Tinnefeld, and Wegener (2005), the authors define jump(i, j) as an operation that causes the element at position i to jump to position j while the elements at positions $i{+}1, \ldots, j$ (if $j > i$) or $j, \ldots, i{-}1$ (if $j < i$) are shifted in the appropriate direction. We keep the same notation in this paper, although this operation is also referred to as *insertion* in other works, such as Baioletti, Milani, and Santucci (2020).

Consider $\sigma \in S_n$, then a pair of indices (i, j) with $i < j$ and $\sigma(i) > \sigma(j)$ is called an *inversion* of σ. The *Lehmer Code* is an alternative way of encoding permutations and it utilizes the concept of inversions, which is why it is also known as the *inversion vector*. The Lehmer code and the Lehmer code space can be defined in various ways. In this paper, we adopt the following definition.

Definition 1. *The Lehmer code space L_n is defined as the Cartesian product $[n] \times [n-1] \times \cdots \times [1]$, and the Lehmer code of any permutation $\sigma \in S_n$ is defined as a sequence of length n, denoted by $L(\sigma) := (L(\sigma)_n, L(\sigma)_{n-1}, \ldots, L(\sigma)_1)$ where*

$$L(\sigma)_{n-i+1} := \#\{j > i \mid \sigma(j) < \sigma(i)\}, i \in [1..n].$$

For example, consider a $\sigma = (3, 5, 4, 1, 2) \in S_5$. Its corresponding Lehmer code is $L(\sigma) = (2, 3, 2, 0, 0)$. By convention, the Lehmer code is indexed from n down to 1. The benefit is that position i has cardinality i. In the remainder of this paper, we may omit the final entry (index 1), as it is always zero by definition. There exists a one-to-one correspondence between the permutation space S_n and the Lehmer code space L_n.

Some other useful definitions are as follows. The n-th Harmonic number is $H_n = \sum_{i=1}^{n} 1/i$, for $n = 1, 2, \ldots$, and $H_0 = 0$. Given an event A, the indicator variable $\mathbf{1}_{\{A\}}$ equals 1 if A occurs and 0 otherwise.

2.2 Mathematical Analysis Tools

Drift theory is about bounding the expected optimization time of a randomized search heuristic by mapping its state space to real numbers and considering the "drift", which is the expected decrease in distance from the optimum in

Algorithm 1: RLS minimizing a function $f : L_n \to \mathbb{R}$ with a given probability vector $\mathbf{p}$ and step operator

Sample $x \in L_n$ uniformly at random
for $t = 1, 2, 3, \ldots$ **do**
  Choose $i \in [2..n]$ according to probability $\mathbf{p}_i$
  $y \leftarrow x$
  $y_i \leftarrow \text{step}_i(x_i)$
  if $f(y) \leq f(x)$ **then** $x \leftarrow y$
end for

a single step. In particular, we use multiplicative and variable drift (see Lengler 2020 for an overview of drift theory). Moreover, we use results on coupon collector processes with non-uniform selection probabilities.

3 Algorithms and Benchmark Functions

Runtime analyses of EAs usually start by considering simple benchmark algorithms on simple, well-structured benchmark functions. Such studies have been successfully performed for the search space $\{0,1\}^n$ of binary representations and enhanced our understanding of increasingly complex EAs (Doerr and Neumann 2020). We adopt this approach by adjusting the commonly studied randomized local search (RLS) and $(1+1)$-EA from binary spaces to the Lehmer code search space as given by Algs. 1 and 2. Both algorithms can be equipped with different *step operators* that decide how the individual mutates at a given position. While the RLS makes a step in exactly one position, the $(1+1)$-EA can make a step in each position $i \in [2..n]$, but with probability $1/(n-1)$.

One may notice that the Lehmer code space is similar to the multi-valued decision variable search space $[r]^n$ considered in theoretical runtime analyses before (Doerr, Doerr, and Kötzing 2018), but with decreasing domain sizes across dimensions. Consequently, we consider the following two *step operators*, adapted from the above-mentioned work:

1. The *uniform* step operator: for $i \in [2..n]$, $\text{step}_i(x_i)$ chooses a value $[0..i-1]\backslash\{x_i\}$ uniformly at random.
2. The ± 1 step operator: for $i \in [2..n]$, $\text{step}_i(x_i) = \max\{0, x_i - 1\}$ with probability $1/2$, and $\text{step}_i(x_i) = \min\{i-1, x_i+1\}$ otherwise.

We refer to RLS equipped with the uniform step operator as RLS with uniform mutation strength, and RLS using the ± 1 step operator as RLS with unit mutation strength. The same naming convention applies to the $(1+1)$-EA. Given that domain sizes differ across dimensions in the Lehmer code space, we assign a probability vector $\mathbf{p}$ to guide the selection of the position to be modified. We consider the following two types of probability vectors:

1. *Uniform probability vector*: Each position $i \in [2..n]$ is selected with the same probability, i.e., $\mathbf{p}_i = 1/(n-1)$.
2. *Proportional probability vector*: The probability of selection for each position $i \in [2..n]$ depends on the domain size at position i, specifically $\mathbf{p}_i = 2(i-1)/(n(n-1))$.

Algorithm 2: $(1+1)$-EA minimizing a function $f : L_n \to \mathbb{R}$ with a given step operator

Sample $x \in L_n$ uniformly at random
for $t = 1, 2, 3, \ldots$ **do**
  for i from n down to 2 **do**
    Set $y_i \leftarrow \text{step}_i(x_i)$ with probability $1/(n-1)$ and set $y_i \leftarrow x_i$ otherwise
  end for
  if $f(y) \leq f(x)$ **then** $x \leftarrow y$
end for

Algorithm 3: permutation-based $(1+1)$-EA minimizing a function $f : S_n \to \mathbb{R}$ with given mutation operator $\text{mut}(\cdot)$

Sample $\sigma \in S_n$ uniformly at random
for $t = 1, 2, 3, \ldots$ **do**
  Choose $k \sim \text{Poi}(1)$
  $\sigma' \leftarrow \text{mut}(\sigma, k)$
  if $f(\sigma') \leq f(\sigma)$ **then** $\sigma \leftarrow \sigma'$
end for

We consider a permutation-based $(1+1)$-EA (see Alg. 3) that operates on the classical linear representation of permutations, which we will refer simply as S_n from now on. The mutation operator is parameterized by an integer k, and is defined via the composition of k elementary operations which are randomly sampled. We investigate the following three mutation schemes:

1. *Transposition*: k transpositions $T_1, T_2, \ldots, T_k$ are selected independently and uniformly at random, and $\text{mut}(\sigma, k) = \sigma \circ T_1 \circ \cdots \circ T_{k-1} \circ T_k$. This operator has been studied in Doerr, Ghannane, and Ibn Brahim (2023).
2. *Adjacent Swap*: the algorithm selects k adjacent transpositions $s_{i_1}, s_{i_2}, \ldots, s_{i_k}$ independently and uniformly at random, and $\text{mut}(\sigma, k) = \sigma \circ s_{i_1} \circ s_{i_2} \circ \cdots \circ s_{i_k}$.
3. *Insertion*: k insertions $\text{jump}(i_1, j_1), \ldots, \text{jump}(i_k, j_k)$ are selected independently and uniformly at random, and $\text{mut}(\sigma, k) = \tau$, where τ is generated by sequentially applying the k insertions on σ. This operator was also studied in Scharnow, Tinnefeld, and Wegener (2005).

The theoretical benchmark functions that we will analyze the algorithms on are based on well-structured benchmarks from the literature, including ONEMAX, LEADINGONES and BINARYVALUE (BINVAL for short) for binary spaces (Droste, Jansen, and Wegener 2002) and the fitness functions derived in Scharnow, Tinnefeld, and Wegener (2005). The simple structure but distinct fitness landscape of these functions illustrates typical optimization scenarios in evolutionary algorithms. Moreover, the functions have been used as building blocks in the design and analysis of more advanced theory-inspired benchmarks in multimodal optimization (Lissovoi, Oliveto, and Warwicker 2023).

We study three groups of benchmark functions based on the above-mentioned ONEMAX, LEADINGONES and BINVAL to conduct the theoretical runtime analyses.

1. $\mathcal{L}$-ONEMAX and INV: $\mathcal{L}$-ONEMAX is a unimodal lin-

ear function defined over L_n where for all $l \in L_n$, $\mathcal{L}$-ONEMAX$(l) = \sum_{i=2}^{n} l_i$. The INV is defined over S_n and for all $\sigma \in S_n$, INV(σ) counts the number of inversions in σ. Our goal is to *minimize* both functions, and the optimum point is $(0, 0, \ldots, 0)$ and the identity permutation, respectively.

2. $\mathcal{L}$-LEADINGZEROS and PLEADINGONES: the two functions are defined over L_n and S_n, respectively. For all $l \in L_n$,

$$\mathcal{L}\text{-LEADINGZEROS}(l) = \max\{i \in [0..n] \mid \forall j \in [n-i+1..n], l_j = 0\},$$

while for all $\sigma \in S_n$,

$$\text{PLEADINGONES}(\sigma) = \max\{i \in [0..n] \mid \forall j \in [1..i], \sigma(j) = j\}$$

Our goal is to *maximize* both functions, thus $f(y) \leq f(x)$ should be replaced by $f(y) \geq f(x)$ in all pseudo-code.

3. FACVAL, NVAL and LEXVAL: The FACVAL is defined over L_n where for all $l \in L_n$,

$$\text{FACVAL}(l) = \sum_{i=2}^{n} (i-1)! \cdot l_i$$

the NVAL is defined over the muti-valued decision variables space $[n]^n$ and for all $x \in [n]^n$,

$$\text{NVAL}(x) = \sum_{i=1}^{n} n^{i-1} \cdot x_i$$

the LEXVAL is defined over S_n. In fact, all permutations $\sigma \in S_n$ can be ordered according to the lexicographic order of their vector representations, indexed from 0. The LEXVAL of a permutation $\sigma \in S_n$ is defined as its rank in this order, e.g., consider $\sigma = (1, 3, 2) \in S_3$, then LEXVAL$(\sigma) = 1$. Our goal is to *minimize* all three functions.

4 Connection Between L_n and S_n

To illustrate the relationships between a Lehmer vector $x \in L_n$ and its corresponding permutation $\sigma \in S_n$, it is crucial to note that the sum of the entries in x equals the number of inversions in σ, which can be proved by definition. Therefore, $\mathcal{L}$-ONEMAX $\circ\, L$ and INV are equivalent functions where L is the bijection from S_n to L_n. The following helper lemma shows that $\mathcal{L}$-LEADINGZEROS $\circ\, L$ and PLEADINGONES, as well as FACVAL $\circ\, L$ and LEXVAL, are equivalent functions. We denote the function $\mathcal{L}$-LEADINGZEROS by LZ and PLEADINGONES by LO for brevity.

Lemma 1. *For any $\sigma \in S_n$, let $L : S_n \to L_n$ denote the bijection which is defined before, then* $\text{LO}(\sigma) = \text{LZ}(L(\sigma))$ *and* $\text{LEXVAL}(\sigma) = \text{FACVAL}(L(\sigma))$.

Moreover, it is well known that applying an adjacent swap to a permutation σ adds or removes exactly one inversion in the resulting permutation (Santucci, Baioletti, and Milani 2015). This makes it particularly interesting to examine how an adjacent swap affects the corresponding Lehmer code representation. The following lemma shows that an adjacent swap affects two entries of the Lehmer code by swapping their entries and changing one of them by ± 1.

Lemma 2. *Given $\sigma, \tau \in S_n$ where $\tau_i = \sigma_{i+1}, \tau_{i+1} = \sigma_i$ and $\tau_j = \sigma_j$ for all $j \in [1..n] \backslash \{i, i+1\}$. Let A denote the event that $L(\sigma)_{n-i+1} \leq L(\sigma)_{n-i}$. Then we have,*

- $L(\tau)_{n-i+1} = L(\sigma)_{n-i} + \mathbf{1}_A$,
- $L(\tau)_{n-i} = L(\sigma)_{n-i+1} - \mathbf{1}_{\overline{A}}$, *and*
- $L(\tau)_{n-j+1} = L(\sigma)_{n-j+1}$ *for all* $j \in [1..n] \backslash \{i, i+1\}$.

Adjacent swaps are of particular interest because they are a proper subset of both (generic) swaps and insertions. Moreover, they allow recreating both insertions and generic swaps as follows. An insertion jump(i, j) is the composition the adjacent transpositions $s_i \circ s_{i+1} \circ \cdots \circ s_{j-1}$ for $j > i$, while a transposition $(i\ j)$ equals jump$(i, j) \circ$ jump$(j-1, i)$ for $j > i$. Analogous expressions hold for $j < i$.

5 Runtime Analysis in Lehmer Code Space

In this section, we rigorously analyze the performance of RLS and the $(1+1)$-EA on several representative benchmark functions introduced in Sect. 3. These analyses provide insights into the behavior of the algorithms under various conditions. In particular, we focus on deriving bounds on the expected number of function evaluations required to reach an optimal solution, also referred to as the expected optimization time.

5.1 RLS with Uniform Mutation Strength

We begin by analyzing RLS equipped with the uniform step operator. For the fitness functions $\mathcal{L}$-ONEMAX or FACVAL, obtaining tight bounds via drift analysis is challenging due to the decreasing dimension sizes in the Lehmer code space. Fortunately, the problem can be reformulated as a variant of the Coupon Collector's Problem with unequal probabilities. The result is stated in the following theorem.

Theorem 1. *Consider* RLS *equipped with uniform step operator and uniform probability vector, then its expected optimization time minimizing either $\mathcal{L}$-ONEMAX or* FACVAL *is bounded from above by $(n-1)^2 \ln n + (n-1)^2$ and from below by $(n-1)^2 \ln n - o(n^2 \log n)$.*

The following theorem provides an exact expression for the expected optimization time of RLS on $\mathcal{L}$-LEADINGZEROS.

Theorem 2. *The expected optimization time of* RLS*, when using the uniform step operator and uniform probability vector on the $\mathcal{L}$-*LEADINGZEROS *function, is $n^3/2 - 2n^2 + nH_n + 3n/2 - H_n$.*

One may notice that there exists an imbalance in the optimization time of the operator across different positions due to the unequal domain sizes in the Lehmer code space. Therefore, it is natural to investigate whether the expected optimization time of RLS will be faster if equipped with the proportional probability vector which, from the probability perspective, smooths out the effects caused by differences in domain sizes. The following two theorems state the results.

Theorem 3. *Consider* RLS *equipped with uniform step operator and proportional probability vector, then its expected optimization time minimizing either $\mathcal{L}$-ONEMAX or* FACVAL *is $n(n-1)(\ln(n) + \Theta(1))/2$.*

Theorem 4. *Consider* RLS *equipped with uniform step operator and proportional probability vector, then its expected optimization time maximizing* $\mathcal{L}$-LEADINGZEROS *is* $n^3/2 - n^2 H_n/2 - n^2/2 + nH_n/2$.

We note that the proportional probability vector gives no asymptotic speed-up compared to the uniform one. Moreover, for the unit step operator, the mutation strength is the same for each position. Therefore, the proportional probability vector will no longer be considered in the subsequent analyses.

5.2 RLS with Unit Mutation Strength

Theorem 5. *Consider* RLS *equipped with* ± 1 *step operator and uniform probability vector, then its expected optimization time minimizing either* $\mathcal{L}$-ONEMAX *or* FACVAL *is* $\Theta(n^2)$.

Theorem 6. *Consider* RLS *equipped with* ± 1 *step operator and uniform probability vector, then its expected optimization time maximizing* $\mathcal{L}$-LEADINGZEROS *is* $2n^4/9 - 7n^3/18 + n^2/9 + n/18$.

5.3 $(1+1)$-EA with Uniform Mutation Strength

Now, we begin to analyze $(1+1)$-EA equipped with uniform step operator, and the following theorem provides a general lower bound for its expected optimization time on both $\mathcal{L}$-ONEMAX and FACVAL.

Theorem 7. *Consider* $(1+1)$*-EA equipped with uniform step operator, then its expected optimization time minimizing either* $\mathcal{L}$-ONEMAX *or* FACVAL *is bounded from below by* $\Omega(n^2 \log n)$.

We state the following two theorems that provide the upper bounds for the expected optimization time on $\mathcal{L}$-ONEMAX and FACVAL which match their lower bound asymptotically.

Theorem 8. *Consider* $(1+1)$*-EA equipped with uniform step operator, then its expected optimization time minimizing* $\mathcal{L}$-ONEMAX *is bounded from above by* $\mathrm{e}(n-1)^2 \ln n + 2\mathrm{e}(n-1)^2 - 2\mathrm{e}(n-1)$.

Theorem 9. *Consider* $(1+1)$*-EA equipped with uniform step operator, then its expected optimization time minimizing* FACVAL *is bounded from above by* $69.2(n-1)^2 \ln n + \mathcal{O}(n^2)$.

Doerr and Pohl (2012) analyze $(1+1)$-EA on all linear functions in the search space $[r+1]^n$. Applying the bound they provide, the expected optimization time of $(1+1)$-EA on NVAL is bounded from above by $\mathcal{O}(n^4 \log\log n)$ and from below by $\Omega(n^2 \log n)$. We improve this bound via the same proof technique used in Theorem 9.

Theorem 10. *The expected optimization time of* $(1+1)$*-EA on* NVAL *is* $\Theta(n^2 \log n)$.

In the following theorem, we provide a tight bound for the expected optimization time of $(1+1)$-EA equipped with uniform step operator on $\mathcal{L}$-LEADINGZEROS.

Theorem 11. *Consider* $(1+1)$*-EA equipped with uniform step operator, then its expected optimization time maximizing* $\mathcal{L}$-LEADINGZEROS *is* $(\mathrm{e}-2)(n-1)^3 + (3-3\mathrm{e}/2)(n-1)^2 + R_n$*, where* $R_n > 0$ *and* $R_n = \mathcal{O}(n \log n)$.

5.4 $(1+1)$-EA with Unit Mutation Strength

The following theorem provides a general lower bound for the expected optimization time of $(1+1)$-EA equipped with ± 1 step operator on both $\mathcal{L}$-ONEMAX and FACVAL.

Theorem 12. *Consider* $(1+1)$*-EA equipped with* ± 1 *step operator, then its expected optimization time minimizing either* $\mathcal{L}$-ONEMAX *or* FACVAL *is bounded from below by* $(n-1)^2$.

The following two theorems respectively provide asymptotic upper bounds for $\mathcal{L}$-ONEMAX and FacVal.

Theorem 13. *Consider* $(1+1)$*-EA equipped with* ± 1 *step operator, then its expected optimization time minimizing* $\mathcal{L}$-ONEMAX *is bounded from above by* $\mathcal{O}(n^2)$.

Theorem 14. *Consider* $(1+1)$*-EA equipped with* ± 1 *step operator, then its expected optimization time minimizing* FACVAL *is bounded from above by* $\mathcal{O}(n^2 \log n)$.

To conclude this section, we provide the exact expression of the expected optimization time on $\mathcal{L}$-LEADINGZEROS in the following theorem.

Theorem 15. *Consider* $(1+1)$*-EA equipped with* ± 1 *step operator, then its expected optimization time maximizing* $\mathcal{L}$-LEADINGZEROS *is* $\frac{32\sqrt{\mathrm{e}}-52}{3}(n-1)^4 + \frac{28-16\sqrt{\mathrm{e}}}{3}(n-1)^3 + \frac{13\sqrt{\mathrm{e}}-12}{36}(n-1)^2 - \frac{\sqrt{\mathrm{e}}}{48}(n-1) - \Theta(1)$.

5.5 Comparison of Runtime Results

Here, we mainly compare the results obtained in this section with previous results based on the canonical representation of permutations. In the seminal work by Scharnow, Tinnefeld, and Wegener (2005), the authors rigorously analyze the performance of a $(1+1)$-EA which chooses $s+1$ ($s \sim \mathrm{Poi}(1)$) operators where each operator is decided by a fair coin flip to be either *jump* or *transposition*. The authors show that the expected optimization time on four fitness functions including INV can be bounded by $\mathcal{O}(n^2 \log n)$ and $\Omega(n^2)$. In the recent work by Baumann, Rutter, and Sudholt (2024), the authors show that the expected time for RLS with *adjacent swap* to optimize INV can be bounded by $\Theta(n^2)$. However, for the general $(1+1)$ EA using the classical representation on INV, the best available upper bound is still $\mathcal{O}(n^2 \log n)$. For comparison, our bound for $\mathcal{L}$-ONEMAX in Theorem 12 and Theorem 13 is $\Theta(n^2)$ which is asymptotically tight.

In the work by Doerr, Ghannane, and Ibn Brahim (2023), the authors show that the expected time for $(1+1)$-EA with *transposition* to optimize PLEADINGONES can be bounded by $\Theta(n^3)$. The same polynomial also appears in Theorem 11 for the related $\mathcal{L}$-LEADINGZEROS, even with exact coefficients for the cubic and quadratic terms. Only for unit mutation (Theorem 15), the Lehmer representation gives a worse complexity, which is due to a random walk behavior.

Due to difficulty and complexity, analyses based on canonical representation are typically given in terms of asymptotic bounds. Therefore, to enable a more detailed comparison, we also include empirical analyses in in the following section.

6 Experimental Analysis

We conduct an experimental investigation with a twofold objective. First, we validate the theoretical runtime analysis on benchmark functions by including the *Harmonic mutation* operator in the comparison, as it has recently become popular in runtime analysis (Doerr, Doerr, and Kötzing 2018; Fischer, Larsen, and Witt 2023). Second, we evaluate the algorithms on real-world instances of the LOP and QAP that, unlike benchmark functions, are NP-hard problems characterized by fitness landscapes that exhibit high multimodality and irregular patterns of neutrality.

The included Harmonic mutation works on the domain $[i]$ by choosing a step size $j \in [1..i-1]$ with probability proportional to $1/j$, and the direction is chosen uniformly at random. Hence, this operator is a compromise between the very local unit mutation and the completely uniform one. Moreover, since RLS search cannot escape local optima, we limit the empirical analysis to $(1+1)$-EAs. In total, we empirically evaluate six algorithms: three operating in the Lehmer space—using harmonic (*Lehmer-Harmonic*), uniform (*Lehmer-Uniform*) and unit (*Lehmer-Unit*) mutation—and three operating in the permutation space—employing the standard jump (*Perm-Jump*), transposition (*Perm-Trans*), and adjacent swap (*Perm-AdjSwap*) mutation.

To validate and complement the theoretical runtime results on benchmark functions, we considered instances of all theoretical benchmark functions with n ranging from 50 to 350. For each algorithm-instance pair, 1000 independent runs are conducted, except for those shown in the third graph of Fig. 1, where 100 runs are performed. Each run reports the number of evaluations required to reach the optimum. Average results are shown in Fig. 1, with equivalent functions grouped together. The graphs show that: (i) at least one Lehmer-$*$ algorithm always outperforms all the Perm-$*$ competitors, and (ii) Lehmer-Harmonic performs well across all the theoretical benchmarks.

To ensure a more meaningful experimental comparison for LOP/QAP, we slightly modified the implementations of the $(1+1)$-EAs so that iterations in which the mutation does not alter the current solution are not counted toward the evaluation budget—a common aspect of algorithm engineering (Pinto and Doerr 2018; Ye, Doerr, Wang, and Bäck 2022). For the sake of space, we also omit Perm-AdjSwap because its performance is not competitive with the others. For the LOP, we consider its minimization variant (equivalent to the original maximization version[1]) in which, given a matrix $\mathbf{B} = [b_{ij}]_{n\times n}$, the objective is to minimize $f(\sigma) = \sum_{i>j} b_{\sigma(i),\sigma(j)}$. In the QAP, an instance is defined by two input matrices $\mathbf{A} = [a_{ij}]_{n\times n}$ and $\mathbf{B} = [b_{ij}]_{n\times n}$, and the goal is to minimize $f(\sigma) = \sum_{i,j} a_{i,j} b_{\sigma(i),\sigma(j)}$. A total of 20 real-world instances have been selected: 10 from the LOP and 10 from the QAP. LOP instances come from the well-established IO benchmark suite (Marti and Reinelt 2022), while QAP instances are from the Skorin-Kapov subset of the QAPLIB benchmark collection (Burkard, Karisch, and Rendl 1997).

Two experiments were conducted: the first follows a fixed-target setting using small instances of size $n = 10$, obtained by subsampling the selected benchmark instances; the second adopts a fixed-budget setting and is performed directly on the original instances, whose sizes range from 42 to 100. In both experiments, each algorithm was executed 1000 times per instance.

In the experiment on instances with $n = 10$, we began by performing an exhaustive search to identify the global optima for all instances. We then executed each algorithm repeatedly with a maximum budget of $1\,000\,000$ evaluations per run. For each run, we recorded whether the optimum was reached, and the corresponding runtime—i.e., the number of evaluations required to reach the optimum, or the maximum budget if the optimum was not attained.

The first graph in Fig. 2 shows the success rates of each algorithm, with results aggregated by problem. To assess algorithmic efficiency, we also considered the empirical runtime measure, defined as the average runtime divided by the success rate, following Wang, Vermetten, Ye, Doerr, and Bäck (2022). This corresponds to the expected number of evaluations required to reach the optimum using a multistart version of the algorithm. Empirical runtimes, aggregated by problem, are shown in the second graph of Fig. 2.

These figures show that Lehmer-Harmonic and Lehmer-Uniform are by far more effective than Lehmer-Unit, both approaching the performance of the Perm-$*$ algorithms in terms of both success rate and empirical runtime. This is particularly evident in the LOP, where their success rates surpass that of Perm-Trans and are very close to that of Perm-Jump.

In the experiment on larger instances, global optima are unknown. Consequently, all runs use the entire evaluation budget, which was set to $1000n$ to keep the computational time manageable. Each run returns the best objective value encountered. For the sake of aggregation, objective values are converted to relative percentage deviations based on the best value observed for each instance. Those relative percentage deviations, aggregated by problem, are presented in the third graph of Fig. 2. The results align with the observations from the small-instances experiment.

7 Conclusion and Discussion

We have studied Lehmer codes, also called inversion vectors, as representations for permutations in EAs. Our main focus was the theoretical runtime analysis of simple EAs using Lehmer codes ("Lehmer-EAs") and a comparison to existing analyses for the classical representation, in particular the seminal work by Scharnow, Tinnefeld, and Wegener (2005). As we show for specific benchmarks, there is a clear correspondence between simple mutations in the Lehmer code and well-known fitness measures like the number of inversions in the classical space. Our runtime analyses are asymptotically tight or even non-asymptotic in most cases and reveal that on most benchmarks, the Lehmer-EAs achieve an expected runtime of $\mathcal{O}(n^2 \log n)$ or $\mathcal{O}(n^2)$, which is on par with the algorithms for the classical representation or even better by a factor of $\Theta(\log n)$. An excep-

[1]Maximizing upper triangular part of the matrix and minimizing lower triangular part induce identical rankings over the solutions.

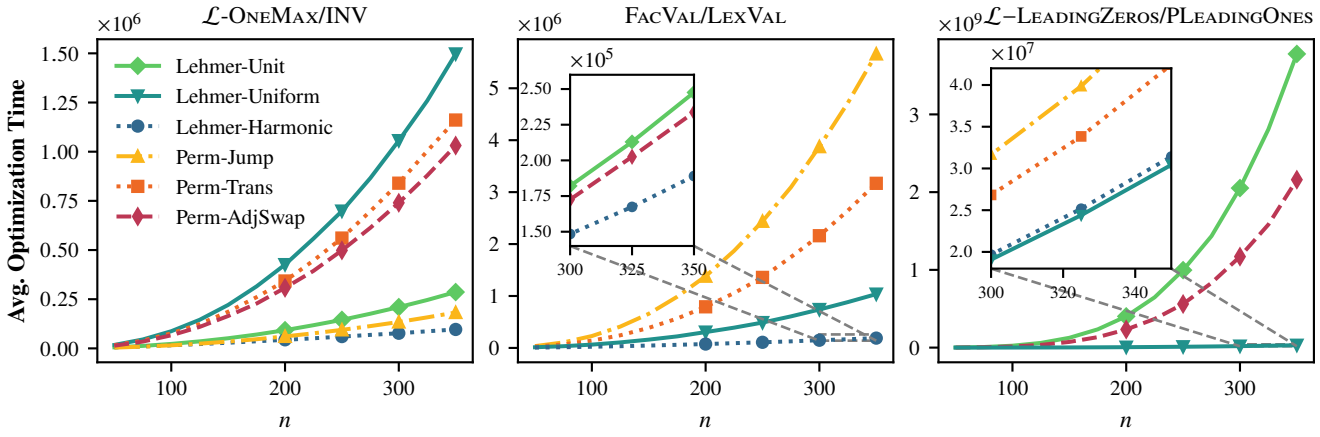

Figure 1: Summary of the results obtained in the experiments on theoretical benchmark functions.

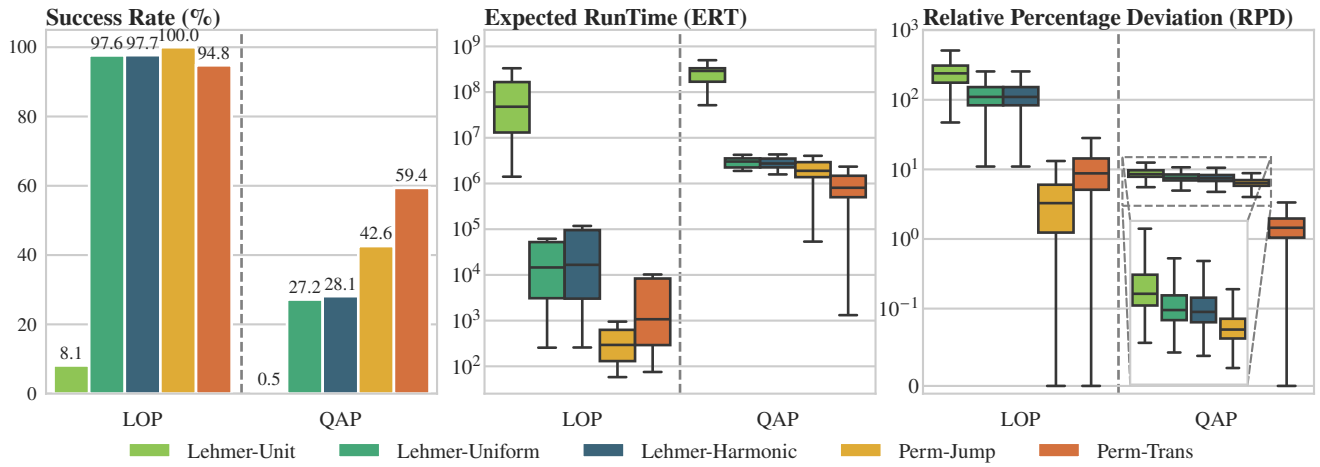

Figure 2: Summary of the results obtained in the experiments on LOP and QAP.

tion is the $\mathcal{L}$-LeadingZeros function, where the expected runtime of the Lehmer-EA using unit mutation is worse by a factor of $\Theta(n)$ than the classical approach. This is due to a random-walk behavior, which can be remedied by the more globally searching uniform mutation.

We supplemented experimental studies of the Lehmer-EAs and compared them to the classical EAs on the theoretical benchmarks and on instances of the linear ordering and quadratic assignment problem. On the theoretical benchmarks, the Lehmer-EA with Harmonic mutation is fastest. While the classical algorithms seem to perform best in general on the empirical benchmarks, the performance of the Lehmer-EAs with Harmonic and Uniform mutation operators are not far behind. In future work, we will analyze whether further runtime improvements for the Lehmer-EAs are possible with advanced operators like self-adjusting mutations and heavy-tailed mutations (Doerr, Doerr, and Kötzing 2018; Doerr, Le, Makhmara, and Nguyen 2017) or by considering variants of the Lehmer code definitions which can be more suitable for optimization purposes (Malagón, Irurozki, and Ceberio 2025).

Finally, although the results presented focus on simple evolutionary algorithms, they may have far-reaching implications, offering valuable insights for the development of more sophisticated methods based on Lehmer codes, as both optimization (e.g. Uher and Kromer 2022) and learning (e.g. Severo, Karrer, and Nolte 2025) have recently begun to explore their use for handling permutations.

Acknowledgements

The third author was supported by the Independent Research Fund Denmark (grant id 10.46540/2032-00101B). Moreover, the research benefited from discussions at Dagstuhl seminar 25092 “Estimation-of-Distribution Algorithms: Theory and Applications”.

References

Baioletti, M.; Milani, A.; and Santucci, V. 2020. Variable neighborhood algebraic differential evolution: An application to the linear ordering problem with cumulative costs. *Information Sciences*, 507: 37–52.

Bassin, A. O.; and Buzdalov, M. 2020. The (1 + (λ, λ)) genetic algorithm for permutations. In *Proceedings Companion of the Genetic and Evolutionary Computation Conference (GECCO '20)*, 1669–1677. ACM Press.

Baumann, J.; Rutter, I.; and Sudholt, D. 2024. Evolutionary Computation Meets Graph Drawing: Runtime Analysis for Crossing Minimisation on Layered Graph Drawings. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO '24)*. ACM Press.

Baumann, J.; Rutter, I.; and Sudholt, D. 2025. Analysing the Effectiveness of Mutation Operators for One-Sided Bipartite Crossing Minimisation. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO '25)*, 872–880. ACM Press.

Bean, J. C. 1994. Genetic algorithms and random keys for sequencing and optimization. *ORSA Journal on Computing*, 6(2): 154–160.

Burkard, R. E.; Karisch, S. E.; and Rendl, F. 1997. QAPLIB–a quadratic assignment problem library. *Journal of Global Optimization*, 10(4): 391–403.

Ceberio, J.; and Santucci, V. 2023. Model-based Gradient Search for Permutation Problems. *ACM Transactions on Evolutionary Learning and Optimization*, 3(4): 1–35.

Doerr, B.; Doerr, C.; and Kötzing, T. 2018. Static and Self-Adjusting Mutation Strengths for Multi-valued Decision Variables. *Algorithmica*, 80(5): 1732–1768.

Doerr, B.; Ghannane, Y.; and Ibn Brahim, M. 2023. Runtime Analysis for Permutation-based Evolutionary Algorithms. *Algorithmica*, 86(1): 90–129.

Doerr, B.; Le, H. P.; Makhmara, R.; and Nguyen, T. D. 2017. Fast genetic algorithms. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO '17)*, 777–784. ACM Press.

Doerr, B.; and Neumann, F., eds. 2020. *Theory of Evolutionary Computation - Recent Developments in Discrete Optimization*. Natural Computing Series. Springer.

Doerr, B.; and Pohl, S. 2012. Run-time analysis of the (1+1) evolutionary algorithm optimizing linear functions over a finite alphabet. In *Proc. of the Genetic and Evolutionary Computation Conference (GECCO '12)*, 1317–1324. ACM Press.

Droste, S.; Jansen, T.; and Wegener, I. 2002. On the analysis of the (1+1) evolutionary algorithm. *Theoretical Computer Science*, 276(1): 51–81.

Fischer, P.; Larsen, E. L.; and Witt, C. 2023. First Steps Towards a Runtime Analysis of Neuroevolution. In *Proc. of Foundations of Genetic Algorithms (FOGA '23)*, 61–72. ACM Press.

Gavenciak, T.; Geissmann, B.; and Lengler, J. 2019. Sorting by Swaps with Noisy Comparisons. *Algorithmica*, (81): 796–827.

Khalil, C.; and Abdou, W. 2021. Transmission of Genetic Properties in Permutation Problems: Study of Lehmer Code and Inversion Table Encoding. In *Proc. of Artificial Intelligence and Soft Computing (ICAISC '21)*, 392–401. Springer.

Lehmer, D. H. 1960. Teaching combinatorial tricks to a computer. In *Proc. of Symposia in Applied Mathematics*, 179–193. American Mathematical Society.

Lengler, J. 2020. Drift Analysis. In Doerr, B.; and Neumann, F., eds., *Theory of Evolutionary Computation: Recent Developments in Discrete Optimization*, 89–131. Springer.

Lissovoi, A.; Oliveto, P. S.; and Warwicker, J. A. 2023. When move acceptance selection hyper-heuristics outperform Metropolis and elitist evolutionary algorithms and when not. *Artificial Intelligence*, 314: 103804.

Malagón, M.; Irurozki, E.; and Ceberio, J. 2025. A combinatorial optimization framework for probability-based algorithms by means of generative models. *ACM Transactions on Evolutionary Learning*, 4(3): 1–28.

Marmion, M.-E.; and Regnier-Coudert, O. 2015. Fitness landscape of the factoradic representation on the permutation flowshop scheduling problem. In *International Conference on Learning and Intelligent Optimization*, 151–164. Springer.

Marti, R.; and Reinelt, G. 2022. *Exact and Heuristic Methods in Combinatorial Optimization*, volume 175. Springer.

Mena, G.; Belanger, D.; Linderman, S.; and Snoek, J. 2018. Learning Latent Permutations with Gumbel-Sinkhorn Networks. In *International Conference on Learning Representations*. OpenReview.net.

Pinto, E. C.; and Doerr, C. 2018. A Simple Proof for the Usefulness of Crossover in Black-Box Optimization. In *Proc. of Parallel Problem Solving from Nature (PPSN '18)*, volume 11102 of *Lecture Notes in Computer Science*, 29–41. Springer.

Regnier-Coudert, O.; and McCall, J. 2014. Factoradic Representation for Permutation Optimisation. In *Proc. of Parallel Problem Solving from Nature (PPSN '14)*, volume 8672 of *Lecture Notes in Computer Science*, 332–341. Springer.

Santucci, V.; Baioletti, M.; and Milani, A. 2015. Algebraic differential evolution algorithm for the permutation flowshop scheduling problem with total flowtime criterion. *IEEE Transactions on Evolutionary Computation*, 20(5): 682–694.

Santucci, V.; Baioletti, M.; and Milani, A. 2020. An algebraic framework for swarm and evolutionary algorithms in combinatorial optimization. *Swarm and Evolutionary Computation*, 55: 100673.

Santucci, V.; and Ceberio, J. 2025. On the use of the Doubly Stochastic Matrix models for the Quadratic Assignment Problem. *Evolutionary Computation*, 33(3): 425–457.

Scharnow, J.; Tinnefeld, K.; and Wegener, I. 2005. The analysis of evolutionary algorithms on sorting and shortest paths problems. *Journal of Mathematical Modelling and Algorithms*, 3(4): 349–366.

Schiavinotto, T.; and Stützle, T. 2007. A review of metrics on permutations for search landscape analysis. *Computers & Operations Research*, 34(10): 3143–3153.

Severo, D.; Karrer, B.; and Nolte, N. 2025. Learning Distributions over Permutations and Rankings with Factorized Representations. *arXiv preprint arXiv:2505.24664*.

Tromble, R.; and Eisner, J. 2009. Learning linear ordering problems for better translation. In *Proc. of the 2009 Conference on Empirical Methods in Natural Language Processing*, 1007–1016. Association for Computational Linguistics.

Uher, V.; and Kromer, P. 2022. Lehmer Encoding for Evolutionary Algorithms on Traveling Salesman Problem. In *Proc. of International Conference on Machine Learning Technologies (ICMLT '22)*, 216–222. IEEE Press.

Wang, H.; Vermetten, D.; Ye, F.; Doerr, C.; and Bäck, T. 2022. IOHanalyzer: Detailed performance analyses for iterative optimization heuristics. *ACM Transactions on Evolutionary Learning and Optimization*, 2(1): 1–29.

Wang, R.; Yan, J.; and Yang, X. 2021. Neural graph matching network: Learning lawler's quadratic assignment problem with extension to hypergraph and multiple-graph matching. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(9): 5261–5279.

Ye, F.; Doerr, C.; Wang, H.; and Bäck, T. 2022. Automated Configuration of Genetic Algorithms by Tuning for Anytime Performance. *IEEE Transactions on Evolutionary Computation*, 26(6): 1526–1538.