Improving genetic algorithms for solving the SVP: focusing on low memory consumption and high reproducibility

Masaharu Fukase1 · Masahiro Kaminaga1

Received: 23 June 2022 / Accepted: 5 September 2022 / Published online: 14 September 2022
© The Author(s) 2022

Abstract
The shortest vector problem (SVP) is absolutely essential in lattice-based cryptography. In this paper, we significantly improve genetic algorithms (GAs) for solving the SVP. GAs, which are simple and powerful optimization techniques, that have the potential to eliminate the limitations of the existing fast SVP algorithms. We improve the entire phase of the GA construction. Our proposed method is based on the concept of low memory consumption and high reproducibility, and this is the main and significant difference between our algorithm and the other SVP algorithms. Our contributions are twofold. First, we developed a new GA for solving the SVP and achieved a considerable improvement in the running time performance. Second, we interpreted certain genetic operations, such as mutation and crossover, in the context of lattices, which has not been done in previous studies. The general result of this paper is that we showed the potential of GAs in the field of lattices.

Keywords Lattice · Shortest vector problem · Genetic algorithm

1 Introduction
A lattice \( L \) is the set of all linear combinations with integer coefficients of a set of linearly independent column vectors \( b_1, \ldots, b_n \in \mathbb{Z}^m \). The integer \( n \) is called the dimension of the lattice \( L \), and \( [b_1, \ldots, b_n] \) is called a basis of the lattice \( L \). A lattice has infinitely many bases that can generate it when \( n \geq 2 \). The problem of finding a nonzero lattice vector of the shortest length for a given lattice basis \( B = [b_1, \ldots, b_n] \) is one of the most studied problems concerning lattices. This problem is called the shortest vector problem (SVP). As the dimension \( n \) increases, the SVP becomes harder.

The SVP, which is the problem we address in this paper, is absolutely essential in lattice-based cryptography, i.e., the security of lattice-based cryptography is based on the hardness of the SVP. Lattice-based cryptography is important because it is a promising candidate for the postquantum cryptography, and SVP algorithms are necessary to assess the security of lattice-based cryptography. Therefore, improving SVP algorithms is very important, and the significance of our results lies in this fact. In this paper, we significantly improve genetic algorithms (GAs) for solving the SVP. GAs, which are simple and powerful optimization techniques, have the potential to eliminate the limitations of the existing fast SVP algorithms.

Many algorithms have been developed to solve the SVP as well as approximate versions thereof. SVP algorithms are necessary to evaluate the security of lattice-based cryptosystems. The most popular benchmark for the performance evaluation of SVP algorithms is the SVP challenge [1], which was first made available in 2010 and has prompted intense competition among many researchers concerning lattices. A 120-dimensional lattice for which Kuo et al. [2] found an optimal solution by parallelizing extreme pruning [3] is an early notable record in the SVP challenge. SVP challenge lattices with dimensions near 120 still seem to be challenging for algorithms performed on a single thread on a single core. Regarding this point, we refer to [4], where it is noted that some higher-dimensional solutions have been achieved using large-scale computational resources.

BKZ 2.0, an improved implementation of the Blockwise Korkine–Zolotarev (BKZ) algorithm developed by Chen and Nguyen [5], was the first to go beyond SVP challenge lattices of dimension 120. Arguably the most notable record was a 126-dimensional lattice solved by Chen and Nguyen...
until Fukase and Kashiwabara [6] began to deliver higher-dimensional solutions using their algorithm, which is a variant of the Random Sampling Reduction (RSR) algorithm [7]. Fukase and Kashiwabara’s RSR variant has delivered an optimal solution for a 128-dimensional lattice. Fukase and Kashiwabara extended the definition of the search space used in [7] by introducing more adjustable parameters and developed a method of generating an advantageous basis based on the reduction of the sum of the squared lengths of the Gram–Schmidt orthogonalized vectors. However, their RSR variant requires very intricate parameter tuning. It requires four array parameters for determining its search spaces and several other atypical parameters, in contrast to the block size parameter $\beta$ of the BKZ algorithm [8]. In addition to this intricate parameter tuning, the much-patched nature of their algorithm and implementation has unintentionally led to insufficient reproducibility of the algorithmic performance. Therefore, it is very difficult to directly follow their technique and reproduce their records. In fact, Aono and Nguyen [4] have implied that the reproducibility of Fukase and Kashiwabara’s algorithmic description itself is low. The method for generating an advantageous basis, which is a crucial part of their algorithm, is especially difficult to reproduce. Although Teruya et al. [9] massively paralleled Fukase and Kashiwabara’s RSR variant, they could not overcome their limitation to a dimension of approximately 150. In summary, their RSR variant reached a stalemate.

It was sieving algorithms that ultimately overcame Teruya and Kashiwabara’s limitation. Sieving algorithms have demonstrated remarkable progress in the past few years. Albrecht et al.’s technique [10] is the best among them. By using this technique, Ducas et al. [11] achieved the current highest record, reaching dimension 180 in the SVP challenge. This record overwhelmingly surpasses that achieved by Teruya and Kashiwabara. Albrecht’s technique uses the orthogonal complement of the linear space spanned by a subset of the Gram–Schmidt vectors of a lattice basis, i.e., a space of a lower dimension than the lattice dimension $n$. The sieving process finds a short vector in the orthogonal complement, which is then lifted to a lattice vector in the lattice by means of Babai lifting [12].

The basic idea of lattice sieving is to obtain shorter lattice vectors through the addition or subtraction of two vectors selected from a set of lattice vectors. As long vectors in the set are replaced with shorter vectors obtained in this way, the lattice vectors in the set become increasingly shorter. Although lattice sieving is powerful, it has a serious drawback: it creates exponential growth in demand for memory resources as the lattice dimension $n$ increases. For example, Ducas et al. [11] reported that they consumed 1.41 TiB out of 1.5 TiB of available RAM ($1 \text{ TiB} = 2^{40} \text{ bytes}$) to reach dimension 180 for the SVP challenge. They also consumed 0.8 TiB and 1.06 TiB for dimensions 176 and 178, respectively. Thus, to achieve higher records, they would definitely need an enormous amount of RAM, much more than 1.5 TiB.

To overcome the drawbacks of the existing fast algorithms, such as RSR variants and sieving algorithms, it is important to focus on GAs. This is because GAs have the potential to overcome the drawbacks noted above. More specifically, they achieve automated parameter optimization and do not impose excessive demands on memory capacity. Automated parameter optimization is important for versatility because in most cases, we must tune some of the parameters of an SVP algorithm (the most typical example is the BKZ parameter $\beta$) when it is applied to unexplored dimensions in a particular lattice or to different types of lattices. Regarding memory consumption, according to our measurement results, GAs require only several to tens of MBs at dimension 180, for example. They also offer high reproducibility because their steps are very simple. Furthermore, Laarhoven [13] investigated the similarities between lattice sieving and evolutionary algorithms (EAs). Since lattice sieving is the most powerful technique for solving the SVP and GAs are a class of EAs, these similarities are of great significance in developing GAs for solving the SVP. Additionally, Harada et al. state in [14] that “genetic algorithms are maybe the most popular class of EAs, and most things done on them can be reproduced in other EAs”. Therefore, there is no reason not to extend GAs to lattices.

The works by Ding et al. [15] and Luan et al. [16] are examples of the application of GAs to the SVP. To obtain an efficient GA for solving the SVP, it is necessary to consider the following basic questions:

1. Which of the existing algorithms is superior?
2. How should we improve the existing algorithms?

The answers remain unclear both because the existing algorithms use different types of chromosomes under different strategies and because a direct comparison among them in the same environment (with the same inputs, the same computer specifications, and programs written under the same conditions) has not yet been performed. In this paper, we present a qualitative and experimental analysis of two existing algorithms to clarify what their advantages and disadvantages are and which of them is more efficient. This is possible because of their reproducibility.

Considering the results of our analysis, we then discuss how we can improve the existing algorithms. In the design of our algorithm, we use a small population size to accelerate the changes over generations. Considering our strategy for solving the SVP, a small population size is a natural choice because it further reduces the memory consumption of the GA. On the other hand, by keeping this important parameter small, we are more likely to miss opportunities to find better chromosomes. To avoid this, we additionally adopt a local
search, adapted to the type of chromosomes that we use. In our mutation and local search operations, we exclude useless chromosomes in advance by making use of the genetic properties of the chromosomes. Furthermore, to generate an advantageous basis in every update (reduction), we use the reproducible method presented in [17]. In our experiments, our algorithm, which is designed as mentioned above, is found to be considerably faster than Ding’s and Luan’s algorithms.

Additionally, we present an interpretation of certain genetic operations, such as mutation and crossover, in the context of lattices. This will be important for the creation a new algorithm in the future. However, Ding et al. and Luan et al. did not develop such an interpretation. In that sense, they treated the lattice as a black box.

The running time of a GA changes significantly in every trial. However, the running time data in [15,16] were not averaged, and hence, the experimental results in [15,16] do not seem to be reliable. In our experiments, we averaged the running time over 100 trials for each dimension; hence, the dimensions covered by our experiments were limited to between 40 and 90. These dimensions are low compared to those in [15,16], where the experiments, whose running time data were not averaged, as stated above, reached dimensions of 118 and 100, respectively. In our future work, we will concentrate on proceeding to higher dimensions.

Even the dimensions of 118 and 100 reached by the existing GAs are low compared to some of the highest records set by existing fast algorithms, especially the overall record of dimension 180, which is the current highest record set by a fast sieving algorithm. However, the existing fast algorithms have been the subject of many studies in the last two decades. For example, sieving algorithms had long been considered impractical since the first description of them was given by Ajtai et al. [18]. However, the utility of algorithms of this type has been enhanced by the emergence of new techniques during the last two decades, and they have recently achieved several of the highest records in the SVP challenge. On the other hand, given their lack of history, GAs for solving the SVP have thus far received little attention. This means that there is plenty of room for progress in improving them to be comparable to the existing fast algorithms. If they could receive sufficient attention, their high reproducibility would greatly assist in such progress.

Furthermore, the existing fast algorithms involve highly advanced parallelization techniques, such as GPU parallelization. On the other hand, importantly, neither Ding’s and Luan’s GAs nor ours have been subjected to any parallelization technique thus far. In [14], where recent advances in parallel GAs are surveyed in detail, Harada and Alba observe that GAs are well suited for parallel execution because in their population-based approach, all solution candidates can be processed in parallel. Therefore, we can be confident that, when subjected to efficient parallelization and performed using sufficient computational resources, genetic algorithms for solving the SVP will be able to proceed to higher dimensions. Since GAs for solving the SVP require only a small memory capacity, we can concentrate on processing units such as CPU cores and GPU units as the computational resources of concern in our future work.

In this study, our research question is as follows: can GAs for solving the SVP be improved? The current knowledge provides no clear sense of direction regarding how they might be improved. In this research, we address this question. Our hypothesis is that improvement is indeed possible. Our solution is not a partial improvement but a total improvement: our improvements span the entire phase of GA construction. Our proposed method is based on the concept of low memory consumption and high reproducibility, and this is the main and significant difference between our algorithm and other SVP algorithms. Our contributions are twofold. First, we develop a new GA for solving the SVP and achieve a considerable improvement in the running time performance. Second, we interpret certain genetic operations, such as mutation and crossover, in the context of lattices, which has not been done in previous studies. The datasets used in our research experiments are standard ones in the field of lattices that are provided at the web link for the SVP challenge [1]. In summary, our general result is that we show the potential of GAs in the field of lattices.

The remainder of this paper is organized as follows. In Sect. 2, we explain some basic concepts regarding lattices and GAs. In Sect. 3, we present a qualitative and experimental analysis of Ding’s and Luan’s algorithms. In Sect. 4, we propose an improved GA for solving the SVP. In Sect. 5, we report a performance evaluation of our algorithm. In Sect. 6, we present an interpretation of our genetic operations in the context of lattices. In Sect. 7, we conclude the paper.

2 Preliminaries

2.1 Basic concepts related to lattices

Given a set of $n$ linearly independent column vectors $B = \{b_1, \ldots, b_n\} \in \mathbb{Z}^{m \times n}$, the integer lattice $L \subset \mathbb{Z}^m$ spanned by $B$ is defined as the set $L(B) = \{Bx \mid x \in \mathbb{Z}^n\}$ of all integral-coefficient combinations of the vectors $b_i$. The integer $n$ is called the dimension of $L$. When $n = m$, we say that $L$ is full-dimensional. The ordered set of vectors $B = \{b_1, \ldots, b_n\} \in \mathbb{Z}^{m \times n}$ is called a basis of $L$. In this paper, we concentrate on full-dimensional integer lattices. A lattice has infinitely many bases that can generate it when $n \geq 2$. Any two lattice bases $B$ and $B'$ generate the same lattice $L$ if and only if there exists a unimodular matrix $U \in \mathbb{Z}^{n \times n}$ (i.e., $\det U = \pm 1$) such that $B' = BU$. The volume of a lattice $L = L(B)$, denoted as
### Algorithm 1 Genetic Algorithm

1. **Encoding**
2. **Initialization**
3. **Fitness evaluation**
4. **repeat**
5. **Selection**
6. **Crossover**
7. **Mutation**
8. **Fitness evaluation**
9. **until** Stopping conditions are met

by \( \text{Vol}(L) \), is defined as the volume of the parallelepiped spanned by the columns of \( B \), i.e., \( \text{Vol}(L) = \text{Vol}(\{ Bx \mid x \in [0, 1]^n \}) \). The volume is a lattice invariant, i.e., it does not depend on any particular basis.

For a lattice basis \( B = \{ b_1, \ldots, b_n \} \), the corresponding **Gram–Schmidt orthogonalized vectors** \( b_1^*, \ldots, b_n^* \in \mathbb{R}^n \) are defined as \( b_i^* = b_i - \sum_{j=1}^{i-1} \mu_{i,j} b_j^* \) with \( \mu_{i,j} = \langle b_i, b_j^* \rangle / \langle b_j, b_j^* \rangle \), where \( \langle x, y \rangle = \sum_{i=1}^{n} x_i y_i \) is the inner product in \( \mathbb{R}^n \). For every \( i, b_i^* \) is the component of \( b_i \) that is orthogonal to \( b_1, \ldots, b_{i-1} \). Consequently, the vectors \( b_1^* \) and \( b_j^* (j \neq i) \) are orthogonal.

We denote the length of the shortest nonzero lattice vector in a lattice \( L \) by \( sh(L) \). A known estimate of \( sh(L) \) is \( (1/\sqrt{n}) \Gamma(n/2 + 1)^{1/n} \) \( \text{Vol}(L)^{1/n} \) [19]. Here, \( \Gamma(n/2 + 1) \) is the Gamma function for \( n/2 + 1 \). In the following, we call the quantity \( (1/\sqrt{n}) \Gamma(n/2 + 1)^{1/n} \) \( \text{Vol}(L)^{1/n} \) the Gaussian heuristic and denote it by \( GH(L) \), following Gama et al. [3] \( GH(L) \) is the radius of the \( n \)-ball whose volume is \( \text{Vol}(L) \). In the SVP challenge, given a lattice basis \( B \), one is asked to find a nonzero lattice vector \( Bx \) such that \( \| Bx \| < 1.05 \cdot GH(L) \).

The Lenstra-Lenstra-Lovász (LLL) algorithm [20] is a key tool for solving the SVP. The LLL algorithm computes a short vector in a lattice in polynomial time. The BKZ algorithm [8] is slower but stronger than the LLL algorithm. The BKZ algorithm is parameterized by a parameter \( \beta \), which determines the search space size of the exhaustive search performed in the algorithm. Although the computational cost increases with larger \( \beta \), the quality of the output basis is also better with larger \( \beta \).

### 2.2 Genetic algorithm

A GA [21] is a probabilistic search algorithm for solving optimization problems. Its operations are based on a simple model of biological evolution. The outline of a simple canonical GA is given in Algorithm 1. A variety of GA variants have been developed following the idea of this canonical GA.

The GA scheme can be summarized as follows.

**Step 1** Represent each of the solution candidates by encoding it as a chromosome (a string). **Step 2** Initialize the algorithm with a population of randomly generated chromosomes. **Step 3** Perform fitness evaluation of all chromosomes in the initial population. **Step 4** Select parents for mating from the current population in accordance with their fitness. **Step 5** Perform crossover between each pair of parents to form a new population. **Step 6** Perform random mutation of the chromosomes in the new population. **Step 7** Perform fitness evaluation of the new chromosomes. **Step 8** Returning to step 4 if the optimal solution (or a near-optimal one) has not been found.

The principle of the survival of the fittest guides the search toward populations containing chromosomes with high fitness values. The mating scheme randomly selects pairs of chromosomes, weighted in favor of chromosomes with higher fitness. Moreover, when replacing old chromosomes in the population with new ones, a GA usually applies an elitist strategy, i.e., ensures that a certain number of chromosomes with the highest fitness values will always survive into the next generation, to prevent the loss of the best found chromosomes.

### 2.3 GAs for solving the SVP

To the best of our knowledge, only three complete algorithms are known thus far that use a GA to solve the SVP. Two of these three algorithms are those of Ding and Luan, while the third was presented by Moghissi and Payandeh [22]. In this paper, we concentrate on Ding’s and Luan’s algorithms. However, our future work will explore the advantages of Moghissi and Payandeh’s algorithm because there are marked and interesting differences between it and the other two algorithms. For example, the former uses the basis vectors of a lattice basis \( B \) as the chromosomes, while the latter two represent a lattice vector as a sequence of integers with a bounded length in relation to the Gram–Schmidt orthogonalized vectors. Moreover, the former utilizes some advanced parallelization techniques, such as GPU parallelization. Since our future work will include parallelizing our algorithm, their study will serve as a useful reference for our research.

In the following, we explain each of Ding’s and Luan’s algorithms.

#### 2.3.1 Ding’s algorithm

Ding’s algorithm is based on \( y \)-sparse representation. A \( y \)-sparse representation is a way to represent a lattice vector as...
a sequence of signed integers with a bounded length. Ding’s algorithm can be summarized as follows.

**Step 1** Generate lattice vectors in the form of the $\gamma$-sparse representation randomly and heuristically.

**Step 2** Represent each of them by encoding it as a chromosome (a bit string), and initialize the algorithm with a population of these chromosomes.

**Step 3** Perform fitness evaluation of all chromosomes in the initial population, where the fitness function is the inverse of the squared Euclidean norm of the lattice vector.

**Step 4** Select parents for mating from the current population, weighted in favor of chromosomes that correspond to shorter lattice vectors.

**Step 5** Perform crossover, random mutation, and a search for a locally optimal chromosome to form a new population.

**Step 6** Preserve the best chromosome, which corresponds to the lattice vector with the shortest length in the last generation of the population.

**Step 7** Perform fitness evaluation of the new chromosomes, where the fitness function is the same one as in step 3.

**Step 8** Return to step 4 if an almost shortest vector has not been found.

2.3.2 Luan’s algorithm

There are two important differences between Ding’s and Luan’s algorithms. First, Luan’s algorithm is based not on the $\gamma$-sparse representation but on the natural number representation (NNR), which was proposed in [6]. NNR has the advantage that a lattice vector in NNR form, in and of itself, can be used as a chromosome because an NNR represents a lattice vector as a sequence of unsigned integers with a bounded length. Furthermore, to avoid local stagnation, Luan’s algorithm adopts a restart strategy under which the current lattice basis is reduced using the best chromosome. Luan’s algorithm can be summarized as follows, from which we can observe some additional differences: for example, Ding’s algorithm employs proportional selection, while Luan’s algorithm employs ranking-based selection to avoid intensive floating-point computations.

**Step 1** Randomly sample lattice vectors in NNR form placed in a restricted candidate set.

**Step 2** Initialize the algorithm with a population of these lattice vectors, which are directly used as chromosomes without any encoding.

**Step 3** Perform fitness evaluation of all chromosomes in the initial population, where the fitness function is the squared Euclidean norm of the lattice vector.

As we will explain further in Sect. 4, Luan et al. showed that the GA can automatically optimize one of the parameters that affect the size of the search space used in [6] although this is not explicitly claimed in [16]. We believe that this is one of the major contributions of Luan et al.’s work.

3 A qualitative and experimental analysis of Ding’s and Luan’s algorithms

In this section, we clarify the advantages and disadvantages of Ding’s and Luan’s algorithms and which of them is more efficient. The individual strengths of the two algorithms are difficult to ascertain without any experiments but might become somewhat clearer when we qualitatively compare their features, as follows:

**Chromosome length for a given dimension** For a given lattice dimension, Luan’s chromosomes are shorter than Ding’s because unlike the former, the latter include entries (bits) for signs and hence are extended by the number of such entries. **Fitness evaluation** Ding’s fitness function is the inverse of the squared Euclidean norm of the lattice vector, while Luan’s is simply the squared Euclidean norm of the lattice vector, which requires a lower computational cost to evaluate than Ding’s function.

**Selection** Ding’s algorithm employs proportional selection. Ding’s proportional selection mechanism computes the distribution of the selection probabilities using the fitness values of all chromosomes and then randomly selects parents for mating in accordance with this distribution. Hence, Ding’s proportional selection mechanism involves intensive floating-point computations. On the other hand, Luan’s algorithm employs uniform random selection. It does not involve floating-point computations because it simply chooses two random integers.
**Mutation** It is known that the initial entries of a short lattice vector in the form of either a \( y \)-sparse representation or an NNR are all 0 (refer to [6,15,16]). This means that because our aim is to find a short lattice vector, we do not need to explicitly consider these initial entries. Moreover, we do not need to mutate any of these initial entries because any such lattice vector of which some of the initial entries are not 0 is unlikely to be short. Nevertheless, both Ding and Luan mutate these initial entries, which does not seem to be efficient.

**Local search** Ding’s algorithm includes a local search procedure that finds a locally optimal chromosome. Ding et al. [15] experimentally justified the effectiveness of his local search technique by noting it accelerated his algorithm by more than a factor of 10 in delivering an optimal solution for a 40-dimensional SVP challenge lattice. In contrast, Luan’s algorithm does not include such a local search procedure.

**Restart strategy** To avoid local stagnation, Luan’s algorithm adopts a restart strategy under which the current lattice basis is reduced using the best chromosome if the best chromosome remains unchanged for a fixed maximum number (e.g., 256) of successive generations. On the other hand, Ding’s algorithm does not adopt such a restart strategy. This means that the initial lattice basis is never updated during the execution of the algorithm, even when it is not advantageous for producing a new chromosome better than the current best one.

**Population size** Ding sets a population size of \( 2n \), where \( n \) is the dimension, while Luan sets a population size of a comparatively large value, e.g., 5000. We have empirically found that when we incorporate a local search procedure into a GA, a large population size such as 5000 increases the running time, whereas a population size of approximately \( 2n \) allows more efficient computation.

In addition to the above qualitative comparison, we have conducted an experimental comparison to ascertain the individual strengths of the two algorithms. In the following, we present a direct experimental comparison between Ding’s and Luan’s algorithms. We implemented our experiments in C++ using the NTL library [22]. All programs were run on a 3.7 GHz Intel Core i9-10900K CPU under Ubuntu 20.04.3 LTS using a single thread on a single core. We measured the running time required for each of the two algorithms to compute an optimal solution for each of the SVP challenge lattices with dimensions of \( n \geq 40 \). For each dimension, we produced a lattice using the generator used in the SVP challenge. We measured the preprocessing time required for the BKZ reduction of the input basis; note that both \( y \)-sparse representation and NNR are valid for a reduced basis (reduced with either the LLL or BKZ algorithm). To ensure fair comparisons, we initialized Ding’s and Luan’s algorithms with the same BKZ parameter \( \beta \) in each dimension; specifically, we set \( \beta = 2\sqrt{n} \), where \( n \) denotes the dimension of the lattice. We determined each of the genetic parameters of Ding’s and Luan’s algorithms, such as population size, crossover rate, and mutation rate, in the same way as in [15,16], respectively. However, our careful reading of the paper by Luan et al. [16] did not allow us to find how to set the total number of offspring in Luan’s algorithm. We therefore set the total number of offspring equal to the number of parents plus the number of elites, which we derived through empirical parameter tuning. For dimensions that Ding’s and Luan’s experiments did not cover, we compensated for the lack of reference genetic parameters by following Ding’s and Luan’s methods of parameter determination, respectively. To enable the reader to replicate our direct comparisons, in Tables 1 and 2, we list all values of all genetic parameters used in these comparisons. In these two tables, \( N \), \( p_{\text{crossover}} \), and \( p_{\text{mutation}} \) denote the population size, crossover rate, and mutation rate, respectively. In Table 2, for simplicity, we omit the first and last entries of \( t \), which is a parameter of NNR, in the same way as in [16]: It follows from Definition 3 in Sect. 4, where a restricted candidate set of lattice vectors in NNR form is defined, that the first and last entries of \( t \) are fixed to 0 and \( n \), respectively.

We see in Fig. 1 that the mean running time of Luan’s algorithm is consistently shorter than that of Ding’s for dimensions larger than 58. In Sect. 5, we compare the results shown in Fig. 1 with those from a performance evaluation due to limited available time. Figure 1 shows, for each of the dimensions considered above, the mean of the running times of Ding’s and Luan’s algorithms. The running time includes the preprocessing time required for the BKZ reduction of the input basis; note that both \( y \)-sparse representation and NNR are valid for a reduced basis (reduced with either the LLL or BKZ algorithm). To ensure fair comparisons, we initialized Ding’s and Luan’s algorithms with the same BKZ parameter \( \beta \) in each dimension; specifically, we set \( \beta = 2\sqrt{n} \), where \( n \) denotes the dimension of the lattice. We determined each of the genetic parameters of Ding’s and Luan’s algorithms, such as population size, crossover rate, and mutation rate, in the same way as in [15,16], respectively. However, our careful reading of the paper by Luan et al. [16] did not allow us to find how to set the total number of offspring in Luan’s algorithm. We therefore set the total number of offspring equal to the number of parents plus the number of elites, which we derived through empirical parameter tuning. For dimensions that Ding’s and Luan’s experiments did not cover, we compensated for the lack of reference genetic parameters by following Ding’s and Luan’s methods of parameter determination, respectively. To enable the reader to replicate our direct comparisons, in Tables 1 and 2, we list all values of all genetic parameters used in these comparisons. In these two tables, \( N \), \( p_{\text{crossover}} \), and \( p_{\text{mutation}} \) denote the population size, crossover rate, and mutation rate, respectively. In Table 2, for simplicity, we omit the first and last entries of \( t \), which is a parameter of NNR, in the same way as in [16]: It follows from Definition 3 in Sect. 4, where a restricted candidate set of lattice vectors in NNR form is defined, that the first and last entries of \( t \) are fixed to 0 and \( n \), respectively.
of our proposed algorithm. The memory consumption for Ding’s and Luan’s algorithms was several MB (e.g., 4.72 MB) and approximately 20 MB (e.g., 17.75 MB) when measured for dimensions 68 and 82, respectively. Thus, in this experiment, both algorithms used only a small amount of memory.

### Table 1 Parameters of Ding’s algorithm: \( l \) denotes the length of a chromosome

| dim | \( N \) | \( p_{\text{crossover}} \) | \( p_{\text{mutation}} \) |
|-----|------|----------------|----------------|
| 40–46 | 5000 | 0.5 | 1/l |
| 48 | 5000 | 0.5 | 1/l |
| 50 | 5000 | 0.5 | 1/l |
| 52 | 5000 | 0.5 | 1/l |
| 52–54 | 5000 | 0.5 | 1/l |
| 56 | 5000 | 0.5 | 1/l |
| 58 | 5000 | 0.5 | 1/l |
| 60–62 | 5000 | 0.5 | 1/l |
| 64–66 | 5000 | 0.5 | 1/l |
| 68–70 | 5000 | 0.5 | 1/l |
| 72–74 | 5000 | 0.5 | 1/l |
| 76–78 | 5000 | 0.5 | 1/l |
| 80–82 | 5000 | 0.5 | 1/l |

However, it is not possible, or at least not appropriate, to simply use Ding’s local search mechanism in Luan’s algorithm. First, Ding’s local search mechanism is specifically designed for lattice vectors in the form of \( y \)-sparse representations; therefore, we need to modify it for application to lattice vectors in NNR form. Second, Ding’s local search mechanism has room for improvement because its search space includes lattice vectors that are unlikely to be short. As stated in Sect. 3, for lattice vectors in either form, we do not need to search lattice vectors of which some of the initial entries are not 0. However, Ding’s local search mechanism does not exclude such vectors. In our local search, we exclude these vectors. Moreover, in both Ding’s and Luan’s algorithms, lattice vectors that are unlikely to be short are also not excluded from mutation. Therefore, we also exclude these vectors in the mutation step.

Luan’s algorithm has another disadvantage. In Luan’s algorithm, the current lattice basis tends to deteriorate with every restart. More specifically, the sum of the squared lengths of the corresponding Gram–Schmidt orthogonalized vectors, i.e., \( \sum_{j=1}^{n} \| b_j \|^2 \), tends to increase with every restart. We refer to the quantity \( \sum_{j=1}^{n} \| b_j \|^2 \) as the G–S sum, following [16]. Fukase and Kashiwabara [6] reported that the smaller the G–S sum is, the more frequently short lattice vectors tend to be found, and presented a heuristic method of reducing the G–S sum. Furthermore, Yasuda et al. [23] theoretically and empirically analyzed the role of the G–S sum in the search for a short lattice vector, and more recently, Yasuda and Yamaguchi [17] developed a polynomial-time algorithm that monotonically reduces the G–S sum, called the \( S^2 \)LLL algorithm. Luan’s restart strategy simply reduces the current lattice basis by means of the LLL algorithm, which cannot keep the G–S sum small. We therefore adopt the \( S^2 \)LLL algorithm to keep the G–S sum small at every restart.

With this reasoning in mind, we improve Luan’s algorithm by adopting an improved local search, a small population size, and the \( S^2 \)LLL algorithm for each restart and by improving Luan’s mutation step. We have empirically confirmed that our algorithm is consistently faster than \( S^2 \)LLL than without \( S^2 \)LLL. We have also confirmed that our algorithm, with or without \( S^2 \)LLL, is considerably faster than Ding’s and Luan’s algorithms. Our algorithm is described in detail in Algorithm 8. Our algorithm uses six genetic subroutines: \textit{Initialization}, \textit{FitnessEvaluation}, \textit{Crossover}, \textit{Mutation}, \textit{LocalSearch}, and \textit{Decode}. In addition, it uses three subroutines for lattice operations: \textit{BKZ}, \textit{ComputeGS}, and \( S^2 \)LLL. We explain each of these subroutines below.

\textit{Initialization} \textit{Initialization} uses the parameter \( t \in \mathbb{N}^{c+1} \) for some \( c \in \mathbb{Z}_+ \) to form the first population. Each chromosome \( z \in \mathbb{N}^n \) is set to a randomly chosen integer in the range \([0, j]\) for \( i = t_j + 1, t_j + 2, \ldots, t_{j+1} \).
FitnessEvaluation FitnessEvaluation computes the squared Euclidean norm of the lattice vector that corresponds to a chromosome $z$. Since $z$ is in NNR form, its norm can be computed from the coefficients $\mu$ of the Gram–Schmidt orthogonalization process and the squared Euclidean norms $c$ of the Gram–Schmidt orthogonalized vectors. CrossoverGiven two parents selected for mating, Crossover forms a new chromosome by randomly selecting a gene at each position from each of these parents.

Mutation Mutation follows Luan’s method except that it leaves each of the initial entries of a chromosome equal to 0, which is our improvement.

LocalSearch LocalSearch is based on the local search mechanism described in [15] and performs a kind of hill-climbing process. Since Ding’s local search mechanism is designed for lattice vectors in the form of $\gamma$-sparse representations, we modify it for application to lattice vectors in NNR form as follows: we assign 1 to $z'[i]$ if $z'[i] = 0$; otherwise, we assign 0 to $z'[i]$. Furthermore, unlike Ding’s local search mechanism, our LocalSearch fixes each of the initial entries to narrow the search space.

Decode Decode transforms a chromosome into a lattice vector in standard form. Decode corresponds to the Generating Algorithm given in [6]. BKZBKZ preprocesses the original basis $B$ using the BKZ algorithm with input parameter $\beta$.

ComputeGS ComputeGS computes the Gram–Schmidt orthogonalized vectors $[b_1^*, \ldots, b_n^*]$ associated with $B$ and assigns the coefficients of the Gram–Schmidt orthogonalization process and the squared Euclidean norms of $[b_1^*, \ldots, b_n^*]$ to $\mu$ and $c$, respectively.

$S^2$LLLOur implementation of $S^2$LLL includes LLL reduction. In each restart run, $S^2$LLL reduces $B$ using the best chromosome and decreases the G–S sum.

In our algorithm, we assign a population to a structure array, each element of which consists of a lattice vector (a chromosome) in NNR form and its norm. We call each element of the array “an individual”. Here, we summarize the key points of our algorithm.

- Our algorithm is based on NNR. We directly use lattice vectors in NNR form as chromosomes.
- Our fitness function is the squared Euclidean norm of the lattice vector.
- We employ uniform random selection.
- We do not mutate any of the initial entries of a chromosome. This is our improvement relative to both Ding’s and Luan’s algorithms.
- Our algorithm includes a local search procedure. Our local search is designed for application to lattice vectors in NNR form. Furthermore, to narrow the search space, it excludes lattice vectors that are unlikely to be short.
- Our algorithm adopts a restart strategy. Unlike Luan’s algorithm, we adopt the $S^2$LLL algorithm, thereby keeping the G–S sum small.
- We set the population size to $2n$.

In the following, we describe how our algorithm works, giving examples of dimension 40. In particular, we demonstrate how each subroutine works, i.e., we present concrete explanations of all of our genetic subroutines.

1. BKZ preprocesses the input basis such that the NNR approach is valid, i.e., any short lattice vector is represented by a sequence of unsigned integers with a bounded length. After BKZ finishes, the following steps are repeated until an optimal solution is found.

2. ComputeGS computes information about the lengths of the lattice vectors. This information is used in FitnessEvaluation, LocalSearch, and Decode.

3. Initialization generates $2 \times 40$ chromosomes. A typical example of a chromosome generated in this step is as follows:

$$z = [0000000000000000000000010001110101]$$

4. FitnessEvaluation evaluates all chromosomes and sorts them by their fitness. Here, the evaluation criterion is the length of the lattice vector.

5. For mating, $2 \times 40$ pairs of parents are randomly selected. Given a pair of parents, Crossover generates an offspring as follows:

$$z_{parent1} = [0000000000000000000000000000001000111020002200]$$

$$z_{parent2} = [000000000000000000000000000000100011100002002020]$$

$$z_{offspring} = [0000000000000000000000000000000000100111020002020]$$

6. Mutation mutates one gene per chromosome on average, as follows:

$$z_{before} = [00000000000000000000000000000000100111020200020202]$$

$$z_{after} = [0000000000000000000000000000000000100111020200021202]$$

7. LocalSearch searches the neighbors of each offspring. When a better chromosome is found, the offspring is replaced with it. A typical neighbor of the offspring in the above example is as follows:

$$z_{offspring} = [0000000000000000000000000000000000000000000100011102020021202]$$

$$z_{neighbor} = [000000000000000000000000000000000000000000010011102020012120]$$

8. The new offspring generated as described above and the elites from the last generation together form a new generation.
9. If the best chromosome in the new generation corresponds to an optimal solution, then our algorithm terminates. If this is not the case, the process returns to step 5. When the number of successive generations with no change in the best chromosome reaches the restart strategy parameter $M$, the process does not return to step 5 but instead advances to the next step.

10. **Decode** transforms the best chromosome in the new generation into a lattice vector in standard form. $S^2$LLL then computes a better lattice basis using this lattice vector and the current lattice basis. The process returns to step 2.

Algorithm 8 formalizes our algorithm and clarifies the cascade relationship between the main routine and its subroutines.

Here, we briefly explain the NNR approach. For the details of NNR, we refer to [6]. The NNR of a lattice vector in $N$ is defined as follows:

**Definition 1 The Natural Number Representation (NNR)**

Let $B$ be a lattice basis, and let $v = Bx$, with $x \in \mathbb{Z}^n$, be a lattice vector in $L(B)$. Given $v = \sum_{j=1}^{n} v_j b_j \in L(B)$ with $v \in \mathbb{R}^n$ such that $v_j = \sum_{i=1}^{n} x_i \mu_{j,i}$, where $b_1, \ldots, b_n \in \mathbb{R}^n$ are the Gram–Schmidt orthogonalized vectors associated with $B$ and $\mu \in \mathbb{R}^{n \times n}$ represents the coefficients of the Gram–Schmidt orthogonalization process, the NNR of $v$ is $z(v) \in \mathbb{N}^n$ such that $-(z_j + 1)/2 < v_j \leq -z_j/2$ or $z_j/2 < v_j \leq (z_j + 1)/2$.

Here, $\mathbb{N}$ denotes the set $\{0, 1, 2, 3, \ldots\}$.

In the experiments in [6], a lattice vector was generated from a restricted candidate set $V_B(s, t)$, which was defined as follows:

**Definition 2 $V_B(s, t) = \{v \in L(B) : d(v) \leq s, w(v) \leq t\}$**

with $d \in \mathbb{N}^n$ and $w \in \mathbb{N}^n$ such that $d_i(v) = \#(z_j(v) : j < i, 1 \leq j \leq n)$ and $w_i(v) = n - \min\{j : z_j(v) = i\} + 1$ for some $c \in \mathbb{Z}_+$, all $i \in \mathbb{N}$ such that $i \leq c$, $s \in \mathbb{N}^c$, and $t \in \mathbb{N}^c$.

Here, $\#S$ denotes the number of elements of a set $S$, and $\mathbb{Z}_+$ denotes the set $\{1, 2, 3, 4, \ldots\}$. We note that $\sum_{i=0}^{c} d_i(v) = n$ for any $v \in L(B)$.

On the other hand, Luan used a simplified candidate set $Z(t)$ of lattice vectors in NNR form that was defined as follows:

**Definition 3 (Luan [16])**

Given a vector $t \in \mathbb{N}^{c+1}$ such that $t_0 = 0 < t_1 < \cdots < t_c = n$, let $Z(t) = \{z \in \mathbb{N}^n : t_j < i \leq t_{j+1}, 0 \leq z_i \leq j, \text{where } j = 0, \ldots, c - 1 \text{ and } i = 1, \ldots, n\}$.

**Algorithm 2 Initialization**

**Algorithm 2 Initialization**

- **Input:**
  - $t$: a parameter of NNR.
  - $N_{\text{population}}$: the parent population size.
  - $N_{\text{offspring}}$: the offspring population size.

- **Global variables defined in the initialization step:**
  - $S_{\text{parent}}$: a structure array, each element of which consists of a lattice vector in NNR form and its norm. $S_{\text{parent}}$ stores all individuals in the parent generation. In the initialization step, some $N_{\text{population}}$ lattice vectors in NNR form are assigned to $S_{\text{parent}}$.
  - $S_{\text{offspring}}$: a structure array, each element of which consists of a lattice vector in NNR form and its norm. $S_{\text{offspring}}$ stores all individuals in the offspring generation.

1: $S_{\text{parent}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{population}})$
2: // assign an array of lattice_vector structures with a size equal to $N_{\text{population}}$ to $S_{\text{parent}}$, where each lattice_vector structure contains a lattice vector (a chromosome) in NNR form and its norm.
3: $S_{\text{offspring}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{offspring}})$
4: // assign an array of lattice_vector structures with a size equal to $N_{\text{offspring}}$ to $S_{\text{offspring}}$
5: for $k := 1, \ldots, N_{\text{population}}$ do
6: for $j := 0, \ldots, \text{t.size}() - 2$ do
7: for $i := [j + 1], \ldots, [j + 1]$ do
8: $S_{\text{parent}}[k].z[i] := \text{RandInt}(1, j) // \text{RandInt}(1, j)$ returns a random integer uniformly distributed in the range $[1, j]$, including both end points
9: end for
10: end for
11: end for

**Algorithm 2 Initialization**

- **Input:**
  - $t$: a parameter of NNR.
  - $N_{\text{population}}$: the parent population size.
  - $N_{\text{offspring}}$: the offspring population size.

- **Global variables defined in the initialization step:**
  - $S_{\text{parent}}$: a structure array, each element of which consists of a lattice vector in NNR form and its norm. $S_{\text{parent}}$ stores all individuals in the parent generation. In the initialization step, some $N_{\text{population}}$ lattice vectors in NNR form are assigned to $S_{\text{parent}}$.
  - $S_{\text{offspring}}$: a structure array, each element of which consists of a lattice vector in NNR form and its norm. $S_{\text{offspring}}$ stores all individuals in the offspring generation.

1: $S_{\text{parent}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{population}})$
2: // assign an array of lattice_vector structures with a size equal to $N_{\text{population}}$ to $S_{\text{parent}}$, where each lattice_vector structure contains a lattice vector (a chromosome) in NNR form and its norm.
3: $S_{\text{offspring}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{offspring}})$
4: // assign an array of lattice_vector structures with a size equal to $N_{\text{offspring}}$ to $S_{\text{offspring}}$
5: for $k := 1, \ldots, N_{\text{population}}$ do
6: for $j := 0, \ldots, \text{t.size}() - 2$ do
7: for $i := [j + 1], \ldots, [j + 1]$ do
8: $S_{\text{parent}}[k].z[i] := \text{RandInt}(1, j) // \text{RandInt}(1, j)$ returns a random integer uniformly distributed in the range $[1, j]$, including both end points
9: end for
10: end for
11: end for

Algorithm 8 formalizes our algorithm and clarifies the cascade relationship between the main routine and its subroutines.

Here, we briefly explain the NNR approach. For the details of NNR, we refer to [6]. The NNR of a lattice vector $v \in L(B)$ is defined as follows:

1. $S_{\text{parent}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{population}})$
2: // assign an array of lattice_vector structures with a size equal to $N_{\text{population}}$ to $S_{\text{parent}}$, where each lattice_vector structure contains a lattice vector (a chromosome) in NNR form and its norm.
3: $S_{\text{offspring}} := \text{Malloc}(\text{lattice}_\text{vector}, N_{\text{offspring}})$
4: // assign an array of lattice_vector structures with a size equal to $N_{\text{offspring}}$ to $S_{\text{offspring}}$
5: for $k := 1, \ldots, N_{\text{population}}$ do
6: for $j := 0, \ldots, \text{t.size}() - 2$ do
7: for $i := [j + 1], \ldots, [j + 1]$ do
8: $S_{\text{parent}}[k].z[i] := \text{RandInt}(1, j) // \text{RandInt}(1, j)$ returns a random integer uniformly distributed in the range $[1, j]$, including both end points
9: end for
10: end for
11: end for

The parameter $t$ used in Definition 3 appears to be slightly different from that used in Definition 2, both variants of $t$ are essentially equivalent: each entry of $t$ determines how large each entry of a lattice vector in NNR form can be and for what range of indices. The simplified candidate set $Z(t)$ is more effective when combined with a GA than $Z(t)$ alone because the GA will automatically optimize the hidden parameters of the candidate set as evolution progresses. On the other hand, a restricted candidate set $V_B(s, t)$ as defined in [6] requires careful parameter selection for $s$, but for the above reason, this seems to be in vain when $V_B(s, t)$ is combined with a GA. Therefore, in our algorithm, we use the simplified candidate set $Z(t)$ rather than $V_B(s, t)$. More specifically, to form the initial population, we use the parameter $t$ used in Definition 3 rather than that used in Definition 2.

We will now explain how the parameter $t \in \mathbb{N}^{c+1}$ for some $c \in \mathbb{Z}_+$ is used to form the initial population. Let $z \in \mathbb{N}^n$ be a chromosome. Each $t_j$, for $j = 0, 1, \ldots, c - 2$, determines how large each $z_j$ can be and for what range of indices: $z_j$ is set to a randomly chosen integer in the range $[0, j]$ for $i = t_j + 1, t_j + 2, \ldots, t_{j+1}$. For example, when $n = 40$ and $t = [0 \ 20 \ 30 \ 40]$, the initial population can include an individual with a chromosome $z = [00000000000000000000111111111122222222222222]$ (this is an extreme example).
5 Performance evaluation of our algorithm

In this section, we present a performance evaluation of our algorithm. We measured the running time of our algorithm for finding an optimal solution for each of the lattices used in the experiments described in Sect. 3. We averaged the running times in the same manner as in Sect. 3. The running time includes the preprocessing time required for the BKZ reduction of the input basis. We initialized our algorithm with the same BKZ parameter β as in Sect. 3; specifically, we set β to $2\sqrt{n}$, where $n$ denotes the dimension of the lattice. Figure 2 shows the mean of the running times for each of the dimensions between 40 and 90, and the data from Fig. 1 are also replotted for comparison.

In the following, we analyze the results.

Running time We see in Fig. 2 that the mean running time of our algorithm is consistently shorter than that of Luan’s algorithm for dimensions larger than 72. For dimensions larger than 72, our algorithm is 11–21 times faster than Luan’s algorithm. We also see that we could not obtain the mean running time of Ding’s algorithm for dimensions larger than 68 and that it is already much slower than both our and Luan’s algorithms at dimension 68. Thus, we can be confi-

![Graph showing running times for different algorithms]
Output:

$s_{1}^{(i)}$: an individual.

$\mu$: the coefficients of the Gram–Schmidt orthogonalization process.

$c$: the squared Euclidean norms of the Gram–Schmidt orthogonalized vectors.

Output:

$s_{1}^{(i)}$: a better individual.

1: for $i := 1, \ldots, n$ do
2: $z'[i] := s_{1}^{(i)} \cdot z[i]$ // the index $i$ decreases from $n$ to 1
3: end for
4: $\text{norm} := \text{FitnessEvaluation}(z', \mu, c)$
5: $\text{is\_found} := 1$
6: while $\text{is\_found} \neq 0$ do
7: $\text{is\_found} := 0$
8: for $i := 1, \ldots, n$ do
9: // narrow the search space by fixing the initial entries in $s_{1}^{(i)}$
10: for $j := 1, \ldots, n$ do
11: $z'[j] := s_{1}^{(i)}[j]$ // the index $j$ decreases from $n$ to 1
12: end for
13: $\text{norm} := \text{FitnessEvaluation}(z', \mu, c)$
14: if $0 < \text{norm} < \text{norm}'$ then
15: for $j := 1, \ldots, n$ do
16: $z'[j] := z[j]$ // the index $j$ decreases from $n$ to 1
17: end for
18: $\text{norm} := \text{norm}'$
19: $\text{is\_found} := 1$
20: end if
21: end for
22: end for
23: for $i := 1, \ldots, n$ do
24: $s_{1}^{(i)}[i] := z'[i]$ // the index $i$ decreases from $n$ to 1
25: end for
26: $s_{1}^{(i)} \cdot \text{norm} := \text{norm}'$
27: end while

dent that our algorithm outperforms both Ding’s and Luan’s algorithms.

**Dimension** Our algorithm reached dimension 90, while Ding’s and Luan’s algorithms could not go beyond dimensions 68 and 82, respectively. We simultaneously started Ding’s, Luan’s, and our algorithms at dimension 40 and increased the dimension in increments of 2. Since we executed 100 trials for each dimension, the differences in the dimensions reached mean that our algorithm completed 1100 and 400 more trials than Ding’s and Luan’s algorithms, respectively, at higher dimensions.

**Memory consumption** The memory consumption for our algorithm was several MB (e.g., 3.81 MB), which was measured at dimension 90. Thus, similar to Ding’s and Luan’s algorithms, our algorithm uses only a small amount of memory. In addition, the memory consumption of our algorithm is smaller than that of Luan’s algorithm (approximately 20 MB at dimension 82). This is natural because we use a much smaller population size.

Table 3 summarizes the results.

Figure 2 also plots the mean running time of our algorithm without S2LLL. We see that it is consistently shorter than that of Luan’s algorithm for dimensions larger than 72, as in the case of our algorithm with S2LLL. Thus, our algorithm, with or without S2LLL, is considerably faster than the previous algorithms proposed by Ding and Luan. We also see that our algorithm is consistently faster with S2LLL than without S2LLL. In summary, we can make the following two statements.

1. Our algorithm outperforms both Ding’s and Luan’s algorithms even without S2LLL.
2. S2LLL is effective in accelerating our algorithm. We can be confident that S2LLL enhances the quality of a restart run.

To enable the reader to replicate our performance evaluation, in Table 4, we list all values of the NNR parameter $t$.
Algorithm 8 Our algorithm

Input:
- $B$: a lattice basis, $[b_1, \ldots, b_n]$.
- $\beta$: the BKZ parameter.
- $t$: a parameter of NNR.
- $M$: the restart strategy parameter.
- $p_{\text{mutation}}$: a mutation rate.

Output:
- $v$: a nonzero lattice vector s.t. $\|v\| < 1.05 \cdot \text{GH}(L(B))$.

1: $\text{BKZ}(B, \beta)$
2: $N_{\text{population}} := \text{Floor}(2.0n)$
3: $N_{\text{elite}} := \text{Floor}(1.0n)$
4: $N_{\text{offspring}} := N_{\text{population}} + N_{\text{elite}}$
5: while $\text{TRUE}$ do
6:   ComputeGS($B, \mu, c$)
7:   Initialization($t$, $N_{\text{population}}, N_{\text{offspring}}$)
8:   for $k := 1, \ldots, N_{\text{population}}$ do
9:       $S_{\text{parent}}[k]$.norm := FitnessEvaluation($S_{\text{parent}}[k], z, \mu, c$)
10:   end for
11:   Sort($S_{\text{parent}}$) // sort $S_{\text{parent}}$ in ascending order of $S_{\text{parent}}[k]$.norm
12:   $z_{\text{best}} := S_{\text{parent}}[1].z$
13:   norm$_{\text{best}}$ := $S_{\text{parent}}[1].$norm
14:   counter := 0
15:   while counter $< M$ do
16:       for $k := 1, \ldots, N_{\text{population}}$ do
17:           repeat
18:               $a := \text{RandInt}(1, N_{\text{population}})$
19:               $b := \text{RandInt}(1, N_{\text{population}})$
20:               $S_{\text{offspring}}[k].z := \text{Crossover}(S_{\text{parent}}[a], S_{\text{parent}}[b])$
21:               $S_{\text{offspring}}[k] := S_{\text{parent}}[k] - N_{\text{population}]$ // preserve $N_{\text{elite}}$
22:               $S_{\text{offspring}}[k] := \text{LocalSearch}(S_{\text{offspring}}[k], \mu, c)$
23:           until $\text{Sum}(S_{\text{offspring}}[k].z) \neq 0$ // avoid a zero vector while allowing the duplication of vectors
24:       end for
25:       for $k := N_{\text{population}} + 1, \ldots, N_{\text{offspring}}$ do
26:           $S_{\text{offspring}}[k] := S_{\text{parent}}[k] - N_{\text{population}]$ // preserve $N_{\text{elite}}$
27:       end for
28:   Sort($S_{\text{offspring}}$) // sort $S_{\text{offspring}}$ in ascending order of $S_{\text{offspring}}[k].$norm
29: if norm$_{\text{best}}$ > $S_{\text{offspring}}[1].$norm then
30:   Decode($S_{\text{offspring}}[1].z, B$) // return an optimal solution
31: end if
32: if norm$_{\text{best}}$ > $S_{\text{offspring}}[1].$norm then
33:     $z_{\text{best}} := S_{\text{offspring}}[1].z$
34:     norm$_{\text{best}}$ := $S_{\text{offspring}}[1].$norm
35:     counter := 0
36: else
37:     counter := counter + 1
38: end if
39: for $k := 1, \ldots, N_{\text{population}}$ do
40:   $S_{\text{parent}}[k] := S_{\text{offspring}}[k]$
41: end for
42: $v := \text{Decode}(z_{\text{best}}, \mu, B)$ // decode $z_{\text{best}}$ into a lattice vector $v$
43: $B := S_{\text{LLL}}(v, b_1, \ldots, b_n)$ // reduce $B$ using $v$ and decrease the G–S sum
44: end while

Table 4 Parameters of our algorithm: $t$ and $M$ denote a parameter of NNR and the restart strategy parameter, respectively.

| dim | $t$       | $M$ |
|-----|-----------|-----|
| 40–46 | [20 30]  | 51  |
| 48   | [25 35]  | 51  |
| 50   | [25 38]  | 51  |
| 52   | [25 40]  | 51  |
| 52–54 | [25 40]  | 51  |
| 56   | [30 40]  | 51  |
| 58   | [31 46 54] | 51  |
| 60–62 | [31 47 55] | 51  |
| 64–66 | [32 48 56 60] | 51  |
| 68–70 | [34 50 58 63] | 64  |
| 72–74 | [36 52 60 64] | 64  |
| 76–78 | [38 45 55 66] | 70  |
| 80–82 | [35 60 70 75] | 70  |
| 84–86 | [35 60 75 80] | 70  |
| 88–90 | [35 65 80 84] | 76  |

Table 4 Parameters of our algorithm: $t$ and $M$ denote a parameter of NNR and the restart strategy parameter, respectively.

and the restart strategy parameter $M$ used in the evaluation. In Table 4, for simplicity, we omit the first and last entries in $t$ in the same way as in Table 2. Although the NNR parameter $t$ specified in Table 4 for each dimension appears to be complicated, it does not require detailed tuning. We roughly determined $t$ such that $S_{\text{parent}}[k].z[i]$ for $i = 1, 2, \ldots, n$ was bounded from above by an increasing integer sequence, and such a $t$ seems to be sufficient to generate a good-quality initial population. By "a good-quality initial population", we mean a population that enables the subsequent populations to converge to an optimal solution within a reasonable running time. Regarding a way to represent a lattice vector as an increasing integer sequence, we refer to [24], the idea of which is associated with the concept of NNR.

In the following, we specify how we selected certain values for the parameters of our algorithm and provide the reasons for our selections.

Crossover parameter We set $p_{\text{crossover}}$ to 0.5. The reason for this parameter selection is that each offspring can evenly inherit the genes of its parents when this parameter has a value of 0.5.

Mutation parameter We set $p_{\text{mutation}}$ to $1/n$. The reason for this parameter selection is that on average, we can mutate one gene per chromosome when this parameter has a value of $1/n$.

Population size We set the population size to $[2.0n]$, as in [15]. The reason for this parameter selection is that a population size of approximately $2n$ seems to allow for efficient computation when we use a local search procedure.

Number of elites We set the number of elites to $[1.0n]$. The reason for this parameter selection is that we need to balance the elitist strategy with genetic diversity: too large a value of this parameter leads to deterioration in genetic diversity, while too small a value causes some promising elites to be missed.
Restart strategy parameter We set the restart strategy parameter $M$, which specifies a fixed maximum number of successive generations with no change in the best chromosome, to a much smaller value than that used in [16]. The reason for this parameter selection is that our algorithm includes a local search procedure, which should accelerate the updating of the best chromosome.

We note that in our algorithm, we parameterize the population size and the number of elites by the lattice dimension $n$, while in Luan’s algorithm, these are not parameterized. Thus, we achieve improvements over Luan’s algorithm in terms of memory consumption, reproducibility, and efficiency.

6 Interpretation of genetic subroutines in the context of lattices

In this section, we present an interpretation of our genetic operations in the context of lattices. This will be important for the creation of a new algorithm in the future. Ding et al. and Luan et al. did not develop such an interpretation. In that sense, they treated the lattice as a black box. In the following, we give an interpretation of our genetic subroutines Initialization, Crossover, Mutation, and LocalSearch in the context of lattices. We omit FitnessEvaluation and Decode here because their interpretation is obvious.

Initialization The chromosomes in NNR form generated by Initialization correspond to lattice vectors that are not too long, yet are not short enough to be candidates for the shortest vector. Each entry of a chromosome is set to a random integer uniformly distributed in some range of integers. On the other hand, the integers of short lattice vectors in NNR form are not uniformly distributed over a certain range of integers; instead, their distribution shows an obvious bias (we refer to [6]). Thus, intuitively, the chromosomes generated by Initialization correspond to lattice vectors that are neither good nor bad. In the subsequent genetic operations, these chromosomes evolve into candidates for the shortest vector.

Crossover In terms of the length of a lattice vector, Crossover generates a chromosome that has the same level of length as its parents. Since the chromosomes from Initialization are neither good nor bad, as stated above, in the initial generations, most of the parents participating in mating are not candidates for the shortest vector, and hence, the optimal chromosome is unlikely to be generated during Crossover. However, as the process of generational change progresses, many chromosomes evolve into candidates for the shortest vector, and hence, there is a greater possibility that the optimal chromosome may be generated during Crossover.

Mutation The most important role of Mutation is to randomly assign values that are not used in Initialization to an NNR with a very small probability. This leads to the possibility of finding an optimal solution whose NNR contains some nontypical values.

LocalSearch For each offspring of crossover and mutation, LocalSearch searches chromosomes whose NNRs are the same as that of the offspring except for a single entry, which is set to either 0 or 1. We adopt this simple assignment method to avoid a large increase in search time.

Considering the above interpretation, there seems to be room for improvement of our genetic subroutines. Accordingly, we present the following improvement plan.

- We may be able to improve the effects of Crossover and Mutation. One possible approach is to adopt some dynamic way of selecting parents for mating. For example, we could select parents through uniform random selection in the initial generations but through proportional selection in more mature generations. Although proportional selection incurs a higher computational cost, it may be worth while to select parents that are weighted in favor of chromosomes corresponding to shorter lattice vectors in mature generations, when there is a greater possibility that the optimal chromosome will be generated during Crossover and Mutation.
- We could extend the search space in LocalSearch by including chromosomes that are missed in our present search. For this extension, we could adopt a more elaborate assignment method in which values other than 0 and 1 are used. However, there is a possibility that this may cause a large increase in search time. Therefore, we will need to consider the tradeoff between search time and search quality.

7 Conclusion

In this paper, we have proposed an improved GA for solving the SVP. In our experiments, our algorithm was considerably faster than the previous algorithms proposed by Ding and Luan. Additionally, we presented an interpretation of several genetic subroutines in the context of lattices. Our plan for future work is to attempt to accelerate our algorithm based on our presented improvement plan, which we developed based on the above interpretation, and to parallelize our algorithm to make use of large-scale computational resources.

Acknowledgements MF discloses support for the research of this work from Japan Society for the Promotion of Science (Grant Number JP20K11816). MK discloses support for the research of this work from Japan Society for the Promotion of Science (Grant Number JP20K11817). The authors thank anonymous reviewers and the editor for their valuable comments and suggestions. The authors also thank Nature Research Editing Service for English proofreading.
**Author Contributions** MF conceived, designed and planned the study, and wrote the main manuscript text, including preparing Figs. 1 and 2. MK participated in the study plan and design, and contributed to data analysis. All authors reviewed the manuscript.

**Funding** This work was supported by Japan Society for the Promotion of Science (Grant Numbers JP20K11816 and JP20K11817).

**Availability of supporting data** The authors confirm that the data supporting the findings of this study are available within the article.

**Declarations**

**Competing interests** The authors have no competing interests as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

**Ethical approval and consent to participate** Not applicable.

**Consent for publication** Not applicable.

**Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article’s Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article’s Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

**References**

1. Schneider, M.: SVP challenge. http://www.latticechallenge.org/svp-challenge/ (2014)
2. Kuo, P.-C., Schneider, M., Dagdelen, Ö., Reichelt, J., Buchmann, J., Cheng, C.-M., Yang, B.-Y.: Extreme enumeration on GPU and in clouds. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 176–191. Springer (2011)
3. Gama, N., Nguyen, P.Q., Regev, O.: Lattice enumeration using extreme pruning. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 257–278. Springer (2010)
4. Aono, Y., Nguyen, P.Q.: Random sampling revisited: lattice enumeration with discrete pruning. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 65–102. Springer (2017)
5. Chen, Y., Nguyen, P.Q.: BKZ 2.0: Better lattice security estimates. In: International Conference on the Theory and Application of Cryptology and Information Security, pp. 1–20. Springer (2011)
6. Fukase, M., Kashiwabara, K.: An accelerated algorithm for solving SVP based on statistical analysis. J. Inf. Process. 23(1), 67–80 (2015)
7. Schnorr, C.P.: Lattice reduction by random sampling and birthday methods. In: Annual Symposium on Theoretical Aspects of Computer Science, pp. 145–156. Springer (2003)
8. Schnorr, C.-P., Euchner, M.: Lattice basis reduction: improved practical algorithms and solving subset sum problems. Math. Program. 61(1), 181–199 (1994)
9. Teruya, T., Kashiwabara, K., Hanaoka, G.: Fast lattice basis reduction suitable for massive parallelization and its application to the shortest vector problem. In: IACR International Workshop on Public Key Cryptography, pp. 437–460. Springer (2018)
10. Albrecht, M.R., Ducas, L., Herold, G., Kirshanova, E., Stevens, M.: The general sieve kernel and new records in lattice reduction. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 717–746 (2019). Springer
11. Ducas, L., Stevens, M., van Woerden, W.: Advanced lattice sieving on GPUs, with tensor cores. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 249–279. Springer (2021)
12. Ducas, L.: Shortest vector from lattice sieving: a few dimensions for free. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 125–145. Springer (2018)
13. Laarhoven, T.: Evolutionary techniques in lattice sieving algorithms. In: International Joint Conference on Computational Intelligence, pp. 31–39. SCITEPRESS-Science and Technology Publications (2019)
14. Harada, T., Alba, E.: Parallel genetic algorithms: a useful survey. ACM Comput. Surv. (CSUR) 53(4), 1–39 (2020)
15. Ding, D., Zhu, G., Wang, X.: A genetic algorithm for searching the shortest lattice vector of SVP challenge. In: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 823–830 (2015)
16. Luan, L., Gu, C., Zheng, Y.: A genetic algorithm with restart strategy for solving approximate shortest vector problem. In: 2020 12th International Conference on Advanced Computational Intelligence (ICACI), pp. 243–250. IEEE (2020)
17. Yasuda, M., Yamaguchi, J.: A new polynomial-time variant of LLL with deep insertions for decreasing the squared-sum of Gram–Schmidt lengths. Des. Codes Cryptogr. 87(11), 2489–2505 (2019)
18. Ajtai, M., Kumar, R., Sivakumar, D.: A sieve algorithm for the shortest lattice vector problem. In: Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing, pp. 601–610 (2001)
19. Schneider, M.: Computing Shortest Lattice Vectors on Special Hardware. Ph.D. thesis, TU Darmstadt (2011)
20. Lenstra, A.K., Lenstra, H.W., Lovász, L.: Factoring polynomials with rational coefficients. Math. Ann. 261(ARTICLE), 515–534 (1982)
21. Holland, J.H.: Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence. MIT Press, Ann Arbor (1992)
22. Shoup, V.: Number Theory C++ Library (NTL), Version 11.4.4. https://www.shoup.net/ntl/
23. Yasuda, M., Yokoyama, K., Shimoyama, T., Kogure, J., Koshiba, T.: Analysis of decreasing squared-sum of Gram–Schmidt lengths for short lattice vectors. J. Math. Cryptol. 11(1), 1–24 (2017)
24. Fukase, M., Yamaguchi, K.: Exhaustive search for finding a very short vector in high-dimensional lattices. In: IWSEC (Short Papers), pp. 26–41 (2010)

**Publisher’s Note** Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.