Double-Structured Genetic Algorithm for Solving the SVP Based on Double Optimization: Using Two Types of Chromosomes

Masaharu Fukase

Tohoku Gakuin University

fukase@mail.tohoku-gakuin.ac.jp

Abstract. The shortest vector problem (SVP) is central in lattice-based cryptography. In this paper, we focus on one of recent classes of algorithms for solving SVP: genetic algorithms (GAs) for lattice. In this paper, we propose the fastest GA to date for solving the SVP. From a search strategy perspective, our algorithm can be seen as an improved version of Fukase's algorithm. On the other hand, from a structure perspective, our algorithm is more than just an improved version of Fukase's algorithm. It is double-structured consisting of: (i) the inner GA, which inherits the basic structure from Fukase's algorithm, and (ii) the outer GA, which optimizes the parameters used in the inner GA. The inner and outer GAs use mutually distinct objective functions. In our algorithm, we use two types of chromosomes and split the criterion for evaluation of chromosomes into two objective functions: the squared Euclidean norm for the inner GA and the evolution velocity of a lattice basis for the outer GA. The latter measures the velocity of the changes of the G-S sum, which is the sum of the squared lengths of the Gram-Schmidt orthogonalized vectors of a lattice basis, and is our novel criterion.

lattice, shortest vector problem, genetic algorithm (GA)

1 Introduction

NIST's standardization process in quantum-resistant public-key cryptography has prompted an increasing amount of research interest in lattice-based cryptography. One of major interests in lattice-based cryptography is security evaluation, where the shortest vector problem (SVP) is central because breaking lattice-based cryptography systems are closely related to this problem. SVP has been one of the most widely studied problems concerning lattices.

A lattice L is the set of all linear combinations with integer coefficients of a set of linearly independent column vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n \in \mathbb{Z}^m$. The integer n is called the dimension of the lattice L, and $[\mathbf{b}_1, \ldots, \mathbf{b}_n]$ is called a basis of the lattice L. A lattice has infinitely many bases that generate the lattice when $n \geq 2$. SVP is the problem of finding a nonzero lattice vector of shortest length for a given lattice basis $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$. As the dimension n increases, SVP gets harder.

Many algorithms have been developed to solve SVP, including its approximate version. The most popular benchmark for performance evaluation of SVP algorithms is the SVP challenge [25], which was opened in 2010 and has prompted intense competition among many researchers concerning lattices. Consequently, the best algorithms to date (i.e., sieving algorithms) have reached dimension 210 for the SVP challenge; it is reported that some preprocessing technique was used, that the max sieving dimension was 171, and that some lifting technique, which constructs the corresponding vectors in higher dimensions from short vectors in lower dimensions, was used. Thus, lattice sieving, which alone can already solve SVP in dimension 171, is overwhelmingly powerful technique for solving SVP, and it draws a lot of attention from researchers in the field of lattices.

As stated in [9], in the SVP challenge web-site, only three types of algorithms (i.e., "enumeration", "sieving", and "BKZ") are explicitly specified and therefore it is significant to submit solutions computed by other new algorithms. In 2015, Ding et al. [9] proposed a GA for solving SVP and demonstrated its effectiveness by applying it to the SVP challenge. More recently, Luan et al. [21] developed another GA for solving SVP based on the technique described in [12]. Furthermore, Fukase and Kaminaga [13] proposed an improved GA for solving SVP based on an analysis and experiment of Ding's and Luan's algorithms, and showed that their GA is the fastest among these three ones.

As stated in detail in [13], the existing fast algorithms such as RSR variants [12][31] (these are the variants of Schnorr's RSR algorithm [28]) and sieving algorithms [5][10][2] [11] (there are many other references which are not cited here) have some problems such as the difficulty in parameter tuning and large memory consumption, and GAs have the potential to overcome these problems. For example, Luan et al. [21] showed that their GA can automatically optimize one of the parameters which affect the size of the search space used in [12]. In addition, Fukase et al. [13] confirmed in their experiments that all the above three GAs for solving SVP use only small memory capacity; they only require several to tens of MBs. As stated in [13], they also provide high reproducibility because the steps of GA, of which the operations are based on a simple model of biological evolution, are very simple.

We note that, in [13], Fukase and Kaminaga did not aim to reach higher-dimensions using their algorithm, but on the other hand they closely evaluated the performances of the above three GAs including their algorithm. Therefore, it remains quite unclear what dimension Fukase's algorithm can solve SVP up to, while that is clear regarding Ding's and Luan's algorithms; Ding's and Luan's ones reached dimension 118 and 100 in [9] and [21], respectively.

In this paper, we propose the fastest GA to date for solving the SVP after demonstrating what dimension Fukase's algorithm can solve SVP up to; we experimentally confirmed that Fukase's algorithm seems not to be able to solve SVP within reasonable running time in dimensions higher than around 108. On the other hand, in our experiments, our algorithm could solve two SVP instances in dimension 120 within a month. We note that we did not use any parallelization technique. This seems to show that our algorithm can be comparable to some of the most popular SVP algorithms, e.g., the Extreme Pruning technique [14]; Using a parallel extreme pruning technique, Kuo et al. [19] computed an optimal solution for a 120-dimensional lattice from the SVP chal-

lenge, which was the highest record as of 2011. SVP challenge lattices of dimension around 120 still seem to be challenging for algorithms performed on a single thread on a single core. Concerning this point, we refer to [3], where it is pointed out that some of higher-dimensional solutions have been achieved using large computational resources.

GAs are definitely able to be further accelerated and to proceed to higher-dimensions using some parallelization on large computational resources because GAs are well suitable for parallel execution as stated in [16]. In fact, as an extension of this study, we have, right up to the present time, tentatively implemented a parallel version of our algorithm simply by file sharing, and this test version over 8 CPU processes could solve SVP in dimension 120 in 4.8 days at its best. Thus, we can be confident that GAs has the potential to be one of the classes of fast algorithms.

From a search strategy perspective, our algorithm can be seen as an improved version of Fukase's algorithm. We integrate two improvement points in our algorithm: avoiding duplication chromosomes and improving the restart strategy. These are discussed in Sections 3 and 4.

On the other hand, from a structure perspective, our algorithm is more than just an improved version of Fukase's algorithm. It is double-structured consisting of: (i) the inner GA, which inherits the basic structure from Fukase's algorithm, and (ii) the outer GA, which optimizes the parameters used in the inner GA. The inner and outer GAs use mutually distinct objective functions, and hence, in this sense, our algorithm is multiobjective (double objective). To our knowledge, we are the first to apply a multiobjective GA to solving the SVP.

While there are trade-offs, which are expressed in Pareto-optimal solutions (also called non-dominated, non-inferior, or efficient solutions), among multiple objective functions in general multiobjective optimization problems, we do not see, at least explicitly, any trade-offs between our two objective functions. This means that our approach is non-Pareto based like Schaffer's vector evaluated GA [24], which is the earliest multiobjective GA and from which many researches that study multiobjective evolutionary algorithms originate.

Our method, however, has a marked difference compared to the conventional ones. The difference is that, in our method, each of objective functions is linked to a different type of chromosome. More specifically, in our algorithm, we use two types of chromosomes; one type and another type are linked to lattice vectors and some parameters, respectively. We then split the criterion for evaluation of chromosomes into two objective functions: the squared Euclidean norm for lattice vectors and the evolution velocity of a lattice basis for the parameters. The latter measures the velocity of the changes of the G-S sum, which is the sum of the squared lengths of the Gram-Schmidt orthogonalized vectors of a lattice basis, and is our novel criterion.

We do not necessarily need well known techniques in the field such as vector evaluation ([24]), where each of objective functions is linked to a subpopulation into which the GA population is randomly divided, due to the difference mentioned above. Therefore, we can extend our algorithm to work with the above two objective functions without destroying the inner structure of it. In our experiments, our double-structured version demonstrated automatic adjustable parameter selection.

In summary, our novelties are:

1. our algorithm is the first double-structured and the fastest GA for solving the SVP to our knowledge and to date, respectively;
2. we are the first to apply a multiobjective GA to solving the SVP;
3. our algorithm uses two types of chromosomes, while previous studies such as Ding et al. and Luan et al. used only one type of chromosome;
4. our outer GA, of which the objective function is our novel criterion "the evolution velocity", realizes automatic adjustable parameter selection, which is also not included in the existing GAs.

The remainder of this paper is organized as follows. In Section 2, we explain some basic concepts of lattices, GAs, and latest GA for Solving SVP. In Section 3, we analyze and improve the problem of duplication chromosomes. In Section 4, we analyze and improve the restart strategy adopted in Fukase's algorithm. In Section 5, we convert our algorithm into a double-structured version to enhance the performance of it by determining optimal values of parameters flexibly. In Section 6, we describe our algorithm using pseudocode and carry out a performance evaluation. In Section 7, we conclude this paper and present future work to be carried out.

We implemented our experiments in C++ using the NTL library [29]. All programs were run on an Intel Core i9-12900K CPU under Ubuntu 22.04.2 LTS. In all of the experiments in this study, we consistently used the SVP challenge lattices which was produced with the random seed set to 0.

2 Preliminaries

2.1 Basic Concepts of Lattices

Given a set of n linearly independent column vectors $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{m \times n}$, the *integer lattice* $L \subset \mathbb{Z}^m$ spanned by B is defined as the set $L(B) = \{B\mathbf{x} \mid \mathbf{x} \in \mathbb{Z}^n\}$ of all integral combinations of $\mathbf{b}_i$'s. The integer n is called the *dimension* of L. When $n = m$, we say that L is *full-dimensional*. The ordered set of vectors $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{Z}^{m \times n}$ is called a *basis* of L. In this paper, we concentrate on full-dimensional integer lattices. A lattice has infinitely many bases that generate the lattice when $n \geq 2$. Any two lattice bases B and B' generate the same lattice L if and only if there is 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 by $\mathrm{Vol}(L)$, is defined as the volume of the parallelepiped spanned by the columns of B, i.e., $\mathrm{Vol}(L) = \mathrm{Vol}(\{B\mathbf{x} \mid \mathbf{x} \in [0,\ 1)^n\})$. The volume is a lattice invariant, i.e., it does not depend on any particular basis.

For a lattice basis $B = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, the corresponding *Gram-Schmidt orthogonalized vectors* $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^* \in \mathbb{R}^n$ are defined by $\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$ with $\mu_{i,j} = \langle \mathbf{b}_i, \mathbf{b}_j^* \rangle / \langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle$ where $\langle \mathbf{x}, \mathbf{y} \rangle = \sum_{i=1}^n x_i y_i$ is the inner product in $\mathbb{R}^n$. For every i, $\mathbf{b}_i^*$ is the component of $\mathbf{b}_i$ that is orthogonal to $\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}$. Consequently, vectors $\mathbf{b}_i^*$ and $\mathbf{b}_j^* (j \neq i)$ are orthogonal.

We can compute $\mathrm{Vol}(L)$ by the product of the lengths of the orthogonalized vectors $\mathrm{Vol}(L) = \prod_{i=1}^n \|\mathbf{b}_i^*\|$ where $\|\mathbf{x}\| = \sqrt{\sum_{i=1}^n x_i^2}$ is the Euclidean norm.

Let $\pi_i : \mathbb{R}^n \to \mathrm{span}(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1})^\perp$ be the orthogonal projection. That is, $\pi_i(\mathbf{v}) = \mathbf{v} - \sum_{j=1}^{i-1} (\langle \mathbf{v}, \mathbf{b}_j^* \rangle / \|\mathbf{b}_j^*\|^2) \mathbf{b}_j^*$.

Algorithm 1 Genetic Algorithm

```
Encoding
Initialization
Fitness evaluation
repeat
    Selection
    Crossover
    Mutation
    Fitness evaluation
until Stopping conditions are met
```

A known estimation of the length of the shortest nonzero lattice vector in a lattice L is $(1/\sqrt{\pi})\Gamma(n/2+1)^{1/n} \cdot (\mathrm{Vol}(L))^{1/n}$ ([26]). Here, $\Gamma(n/2+1)$ is the Gamma function for $n/2+1$. In the following, we call the amount $(1/\sqrt{\pi})\Gamma(n/2+1)^{1/n} \cdot (\mathrm{Vol}(L))^{1/n}$ Gaussian Heuristic and denote it by $GH(L)$ following [14] et al. $GH(L)$ is the radius of the n-ball whose volume is $\mathrm{Vol}(L)$. In the SVP challenge, given a lattice basis B, one is asked to find a nonzero lattice vector $B\mathbf{x}$ such that $\|B\mathbf{x}\| < 1.05 \cdot GH(L)$.

The LLL algorithm [20] is a key tool for SVP. The LLL algorithm computes a short vector in a lattice in polynomial time. The BKZ algorithm [27] is slower but stronger than the LLL algorithm. The BKZ algorithm is parameterized by a parameter β, which determines the search space size of the exhaustive search inside BKZ. Although the computational cost increases for larger β, the quality of the output basis is better for larger β.

2.2 Genetic Algorithm

The genetic algorithm (GA) [17] 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 described 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 Representing each of solution candidates by an encoding as a chromosome (a string).
Step 2 Starting the algorithm with a population of randomly generated chromosomes.
Step 3 Fitness evaluation of all chromosomes in the first population.
Step 4 Selecting mating parents from the current population according to their fitness.
Step 5 Crossover between each parent pair to form a new population.
Step 6 Random mutation of chromosomes in the new population.
Step 7 Fitness evaluation of new chromosomes.
Step 8 Returning to step 4 if the optimal solution (or a near-optimal one) is not found.

The survival of the fittest principle guides the search towards populations containing chromosomes with a high fitness value; The mating scheme randomly selects two chromosomes, weighted in favor of chromosomes with higher fitness. Moreover, in the replacement of old chromosomes by new ones, the GA usually applies the elitist strategy,

i.e., a certain number of the chromosomes with the highest fitness values always survive into the next generation, so as to prevent the loss of the best found chromosomes.

2.3 Latest GA for Solving SVP

To our best knowledge, only three complete algorithms are known so far which use the GA to solve SVP and which are faster than the classical BKZ algorithm. Two of the three algorithms are Ding's and Luan's ones ([9] and [21], respectively), and the third is Fukase's one ([13]). In this section, we basically concentrate on Fukase's algorithm because it was already shown in [13] that Fukase's one is the fastest among these three algorithms.

In [13], Fukase and Kaminaga performed a direct comparison among the three algorithms using the SVP challenge lattices in the same environment. In the comparison experiment, the mean running times were measured for each of the dimensions between 40 and 90 (the dimension was increased in increments of 2) and were reported to be consistently the shortest with Fukase's algorithm for dimensions larger than 72. However, it has not been shown yet what dimension Fukase's one can solve SVP up to, while Ding's and Luan's ones reached dimension 118 in about 3–4 months and 100 in about 3 days in [9] and [21], respectively. Therefore, Ding's dimension 118 has been the highest record to date in GAs for solving the SVP.

Fukase's algorithm is based on the natural number representation (NNR), which was proposed in [12]. Here, we briefly explain the natural number representation (NNR). For the details of NNR, we refer to [12].

Definition 1. ***The Natural Number Representation (NNR)*** *Let B be a lattice basis, and let $\mathbf{v} = B\mathbf{x}$ with $\mathbf{x} \in \mathbb{Z}^n$ be a lattice vector in $L(B)$. Given $\mathbf{v} = \sum_{j=1}^{n} \nu_j \mathbf{b}_j^* \in L(B)$ with $\boldsymbol{\nu} \in \mathbb{R}^n$ such that $\nu_j = \sum_{i=1}^{n} x_i \mu_{i,j}$, where $\mathbf{b}_1^*, \ldots, \mathbf{b}_n^* \in \mathbb{R}^n$ are the Gram-Schmidt orthogonalized vectors associated to B and $\boldsymbol{\mu} \in \mathbb{R}^{n \times n}$ is the coefficients of the Gram–Schmidt orthogonalization process, the natural number representation of $\mathbf{v}$ is $\mathbf{z}(\mathbf{v}) \in \mathbb{N}^n$ such that $-(z_j + 1)/2 < \nu_j \leq -z_j/2$ or $z_j/2 < \nu_j \leq (z_j + 1)/2$.*

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

In [13], Fukase and Kaminaga used a restricted candidate set $Z(\mathbf{t})$ of NNRs, which is defined as follows:

Definition 2. *(Luan[21]) Given a vector $\mathbf{t} \in \mathbb{N}^{c+1}$ such that $\mathbf{t}_0 = 0 < \mathbf{t}_1 < \ldots < \mathbf{t}_c = n$, let $Z(\mathbf{t}) = \{\mathbf{z} \in \mathbb{N}^n : \text{for } \mathbf{t}_j < i \leq \mathbf{t}_{j+1}, 0 \leq \mathbf{z}_i \leq j, \text{where } j = 0, \ldots, c-1 \text{ and } i = 1, \ldots, n\}$.*

We note that NNRs placed in $Z(\mathbf{t})$ do not directly include information about the lattice basis or the corresponding lattice vector; it is not until a lattice basis B is given that these NNRs are linked to the corresponding lattice vectors in standard form.

Fukase's algorithm can be summarized as follows.

Step 1 Randomly sampling NNRs in $Z(\mathbf{t})$.

Step 2 Starting the algorithm with a population of these NNRs, which are directly used as chromosomes without any encoding.

Step 3 Fitness evaluation of all chromosomes in the first population, where the fitness function is the squared Euclidean norm of a lattice vector.

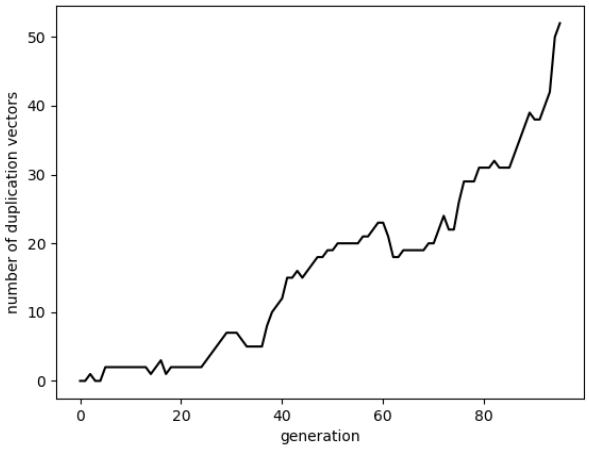

Fig. 1: The transition of the number of the duplication chromosomes for the 120-dimensional SVP challenge lattice

Step 4 Randomly selecting mating parents from the current population.

Step 5 Crossover, random mutation, and search of a locally optimal chromosome for each parent pair. During local search, fitness evaluation of the new chromosomes are also performed.

Step 6 Ranking-based selection with elitist strategy, where some of the best chromosomes produced by crossover, mutation, and local search form a new population along with some of the best ones in the last generation of population.

Step 7 Terminating the algorithm if an optimal solution is found. Otherwise returning to step 4.

Step 8 Reducing the current lattice basis using the best chromosome and returning to step 1, which we call a restart run, if the best chromosome remains unchanged for a long time (for a fixed maximum number of successive generations).

3 Avoiding Duplication Among Chromosomes

We examined the state of chromosomes in each generation during the execution of Fukase's algorithm. We then found out that the duplication of chromosomes occurs more frequently as the generation proceeds. Figure 1 shows the transition of the number of the duplication chromosomes between the beginning of the execution and the first update reduction in a search, where the population size is 240, for the 120-dimensional SVP challenge lattice. We mean the number of chromosomes whose gene array is the same with that of at least one other chromosome in the same generation by the number of the duplication chromosomes. We experimentally confirmed that in general cases

for the 120-dimensional SVP challenge lattice the transition of the number of the duplication chromosomes between update reductions has the same tendency with Figure 1 and that a case where the number of the duplication chromosomes gets close to the population size is not unusual. It is widely known that in GAs lack of population diversity commonly leads to stagnation during evolution process, and hence we should avoid such a case.

In the Step 6 of Fukase's algorithm, the duplication of chromosomes is allowed, while zero vectors are removed. Our observation suggests that the duplication of chromosomes should also be avoided. We can avoid duplication among chromosomes by using a property of ranking-based selection as in the next paragraph.

In Fukase's ranking-based selection, all candidate chromosomes for the new generation is sorted in ascending order of the norm of the corresponding lattice vector, and the same number of candidate chromosomes counted from the start with the population size are passed on to the new generation. To avoid the duplication of chromosomes, every time we encounter a candidate chromosome that is supposed to be passed on to the new generation and that has the same norm as the previous candidate chromosome that has already been passed to the new generation, we remove it. If there are too many duplication chromosomes and as a result the number of the survived candidate chromosomes does not reach the population size, we then call a restart run and initialize all chromosomes because we can judge that diversity has been lost in that case. This process of avoiding the duplication of chromosomes corresponds to the code from Line 70 to Line 82 of Algorithm 2. In our following experiments, we always use this process.

4 Improvements on the Restart Strategy

For a restart run, Fukase's algorithm uses the S^2LLL algorithm, of which Fukase's implementation includes LLL reduction that can be replaced with BKZ reduction, to decrease the G-S sum. As stated in Section 2.3, we ran 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice, and in all of them an optimal solution was not found in 6 months. To identify the causes of this limitation of Fukase's algorithm, we examined the changes of the G-S sum in the above trials. We show in Figure 2 the change of the G-S sum during one of the above trials (in about 6 months). Figure 2 also shows the change of the squared Euclidean norm of the shortest lattice vector ever found in the right axis. We see in Figure 2 that the G-S sum hardly decreases in a long period of execution except for the first few period and that the squared Euclidean norm also hardly decreases in conjunction with the G-S sum. We examined whether this holds true for each of the above trials and observed similar results in all of them. We confirmed in all of the 10 trials that the G-S sum had never come down to below 3.0×10^8 in 6 months.

In the following subsections, we show a method to further decrease the G-S sum and how to manage a stagnation which seems to be difficult to be overcome by the method alone.

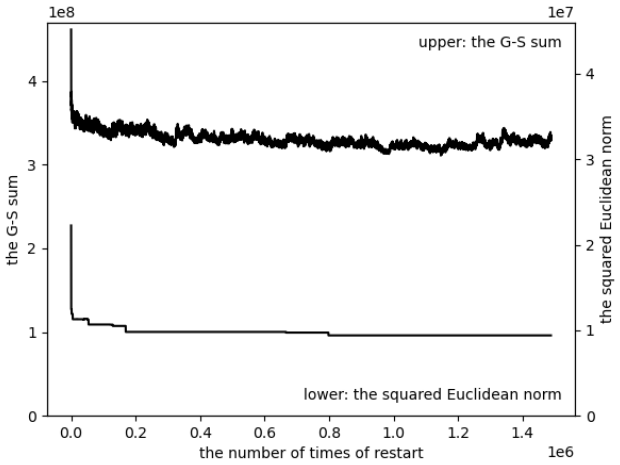

Fig. 2: The change of the G-S sum (the left axis) and the change of the squared Euclidean norm of the shortest lattice vector ever found (the right axis) during one of 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice.

4.1 Decreasing the G-S sum by Using the Insertion Chromosome

We use a specific chromosome in the reduction for a restart run to further decrease the G-S sum. For that, we define *the insertion chromosome* ch_{in} as follows.

Definition 3. *Let B be the lattice basis between some two update reductions, and let $[\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$ be the Gram-Schmidt orthogonalized vectors corresponding to B. The insertion chromosome ch_{in} is a chromosome in one of successive generations between that time period such that the corresponding lattice vector $\mathbf{v_{in}}$ satisfies $\mathbf{v_{in}} \in \arg\max_{\mathbf{v}\in V_B}(\|\mathbf{b}_l^*\|^2 - \|\pi_l(\mathbf{v})\|^2)$ where V_B is a set of corresponding lattice vectors of all chromosomes in successive generations between that time period and $l = \min_{1\le j\le n}\{j \mid \|\pi_j(\mathbf{v})\|^2 < \|\mathbf{b}_j^*\|^2\}$.*

By definition, there can exist more than one ch_{in}, but we only use the first occurrence of ch_{in} in our algorithm. We can decrease $\|\mathbf{b}_j^*\|^2$ for some j at the possible maximum extent of decrease between the two update reductions by the insertion of ch_{in} in B. For a restart run, the corresponding lattice vector $\mathbf{v_{in}}$ of ch_{in} is inserted to B, and the generating system consisting of B and $\mathbf{v_{in}}$ is reduced by S^2LLL. $\mathbf{v_{in}}$ is inserted at index l of B with l described in Definition 3. Definition 3 almost guarantees by the restriction on the index l that $\mathbf{b}_j^*$ for $j < l$ is unchanged during the update reduction.

We show in Figure 3 the change of the G-S sum during a search where ch_{in} is used for a restart run in the 120-dimensional SVP challenge lattice in about 3 months. Figure 3 also shows the change of the squared Euclidean norm of the shortest lattice vector ever found in the right axis. We see in Figure 3 the same tendency with Figure 2, but there is

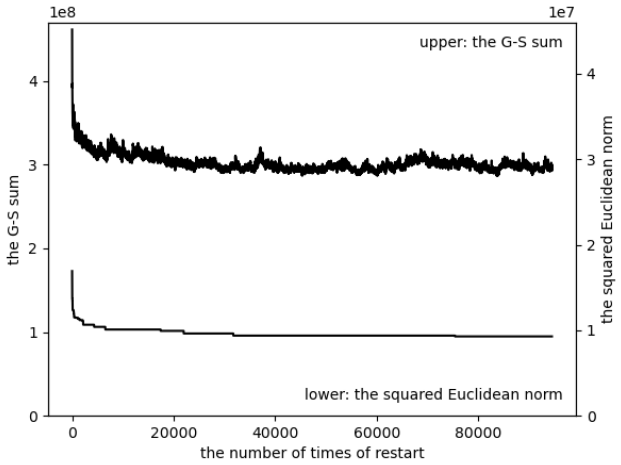

Fig. 3: The change of the G-S sum (the left axis) and the change of the squared Euclidean norm of the shortest lattice vector ever found (the right axis) during a search where ch_{in} is used for a restart run in the 120-dimensional SVP challenge lattice.

a significant difference; in Figure 3, the G-S sum comes down to below 3.0×10^8. We saw in several searches where ch_{in} is used for a restart run in the 120-dimensional SVP challenge lattice including the case of Figure 3 that the G-S sum came down to below 3.0×10^8 in less than 3 months (in less than a month in the case of Figure 3).

We thus confirmed that the G-S sum can be further decreased using the above method (i.e., using the insertion chromosome ch_{in}). Fukase and Kashiwabara presented a different method to decrease the G-S sum in [12]. Specifically, the method in [12] uses the insertion index $h(\mathbf{v})$ of $\mathbf{v}$, which is defined for some δ as in Definition 4, and has several distinct differences against ours as follows.

1. Their method restricts the index of vectors in the basis where a short candidate vector can be inserted for some integer $1 \leq l \leq n$, which is called the restriction index. It does not insert $\mathbf{v}$ with $h(\mathbf{v}) < l$ for the restriction index l and decrease the G-S sum from the front by increasing l; l is increased when the number of times that $\mathbf{b}_l$ is not changed reaches a constant c, which is determined in advance.
2. Their method does not necessarily prioritize the extent of decrease in $\|\mathbf{b}_j^*\|^2$; It preferentially decreases $\|\mathbf{b}_l^*\|^2$ even if $\|\mathbf{b}_j^*\|^2$ can be more decreased at some index $j > l$.

Definition 4. *([12]) Let B be a lattice basis, $\mathbf{v}$ be a lattice vector in $L(B)$, and $\delta \in \mathbb{R}$ with $\delta \leq 1$. The insertion index $h(\mathbf{v})$ of $\mathbf{v}$ is j such that* $\min\{j : \|\pi_j(\mathbf{v})\|^2 < \delta\|\mathbf{b}_j^*\|^2, n+1\}$.

Before we explain the differences between the decreasing method in [12] and our method, we examine how the G-S sum is decreased by our method in the following. We show in Figure 4 the initial change of the index where ch_{in} is inserted during a search

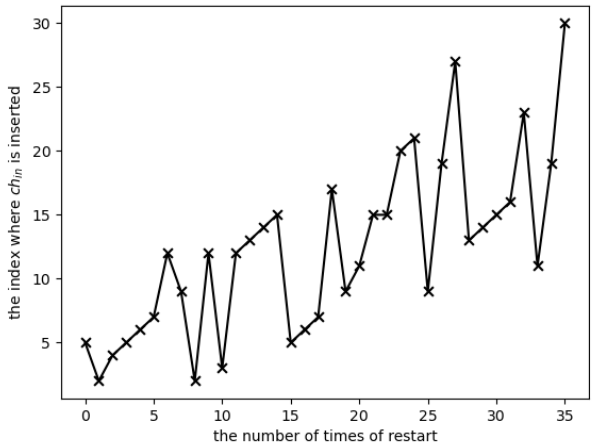

Fig. 4: The initial change of the index where ch_{in} is inserted during a search in the 120-dimensional SVP challenge lattice.

in the 120-dimensional SVP challenge lattice. We can find a pattern in Figure 4. The pattern is that the index is consecutively increased by 1 at several places in Figure 4. This pattern seems somewhat artificial, but we can explain why the index moves in this way as in the following paragraph.

As we explained in Section 2, $\mathrm{Vol}(L) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|$ is a lattice invariant. This means that, if $\|\mathbf{b}_j^*\|^2$ is decreased for some j, $\|\mathbf{b}_k^*\|^2$ is inevitably increased for some ks such that $k \neq j$. Since Definition 3 almost guarantees that $\mathbf{b}_k^*$ for $k < j$ is unchanged during the update reduction, $\|\mathbf{b}_k^*\|^2$ is certainly increased for some ks such that $k > j$ in that case. We here recall that Schnorr's geometric series assumption (GSA) [28] states $\mathbf{b}_j^*$ gets shorter as j increases. GSA states, in other words, $\|\mathbf{b}_k^*\|^2$ with $k = j+1$ is closest to $\|\mathbf{b}_j^*\|^2$ for $k > j$, and hence $\|\mathbf{b}_{j+1}^*\|^2$, especially, is likely to be increased so that $\mathrm{Vol}(L) = \prod_{i=1}^{n} \|\mathbf{b}_i^*\|$ is fixed to be a constant. $\|\mathbf{b}_{j+1}^*\|^2$ is then also likely to be decreased at the next restart because $\|\mathbf{b}_{j+1}^*\|^2$ is likely to be increased at the previous restart and hence tends to be relatively large, and consequently the above pattern is likely to occur frequently.

In Figure 5, which shows the change of the index up to the 200th restart, we can see another pattern. The pattern is that the index roughly keeps on increasing and decreasing. For example, we see that the index increases from around the 50th restart, that it rapidly decreases at around the 125th restart, and that it then increases again at around the 150th restart.

We expect that the above explanation holds true better when $\|\mathbf{b}_j^*\|^2$ is more decreased. This is the reason that we decrease $\|\mathbf{b}_j^*\|^2$ at the possible maximum extent of decrease. For a restart run, we use the restart strategy parameter M, which is a fixed maximum number of successive generations, as in [13]. For larger M, each interval be-

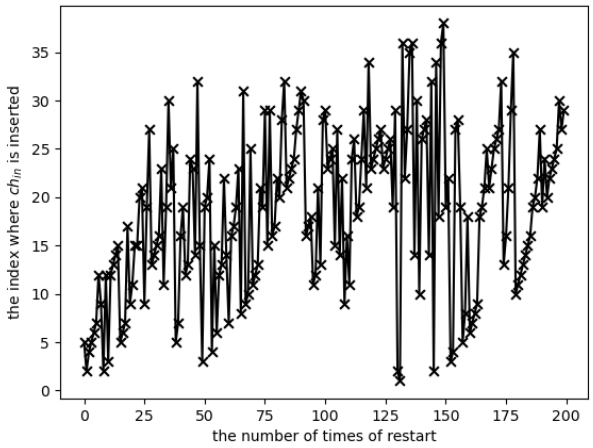

Fig. 5: The change of the index where ch_{in} is inserted up to the 200th restart during a search in the 120-dimensional SVP challenge lattice.

tween update reductions gets longer, and hence a better ch_{in} which more decreases $\|\mathbf{b}_j^*\|^2$ becomes likely to be found. We show in Figure 6 the initial change of the index where ch_{in} is inserted during a search in the 120-dimensional SVP challenge lattice with 10 times larger M than in Figures 4 and 5. In Figure 6, we see more clearly the above two patterns. This seems to support our idea that we decrease $\|\mathbf{b}_j^*\|^2$ at the possible maximum extent of decrease.

Regarding the above two patterns, our method roughly simulates the behavior of the decreasing method in [12], where the index also keeps on increasing and decreasing. The decreasing method in [12] seems to be somewhat inefficient. For example, it requires for both the limit of the insertion index and the limit of the number of times of the insertion at each index to be determined manually. Although these limits need to be customized in every dimension, any method to determine the optimal values of these limits is not presented in [12]. Furthermore, the decreasing method in [12] uses several lattice bases at the same time and stores many lattice vectors in the memory. Since we want to realize an efficient method with low memory consumption as in [13], the decreasing method in [12] seems not to fit in our algorithm. The important thing is that our method produces the effect of the decreasing method in [12] in a more efficient manner than it.

4.2 Controlling the Insertion Index

Although we can decrease the G-S sum as we expected by using the method presented in the previous subsection, it gets harder to decrease $\|\mathbf{b}_j^*\|^2$ in the front indices as the number of times of update reduction increases; it is harder to decrease $\|\mathbf{b}_j^*\|^2$ which is

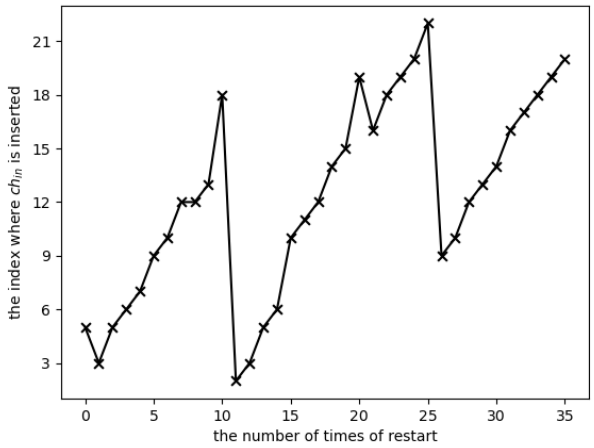

Fig. 6: The initial change of the index where ch_{in} is inserted during a search in the 120-dimensional SVP challenge lattice with 10 times larger M than in Figures 4 and 5.

already decreased to some extent in the front indices than in the back indices. This tends to cause a stagnation, by which we mean a state that the current shortest lattice vector has not been updated in a while; we saw examples of this in Figures 2 and 3. To avoid the stagnation, we introduce two measures; (i) the average of the index of insertion and (ii) the ratio of the number of times of insertion in the relative back indices to the one in the relative front indices. We control the insertion index based on these two measures.

We note that different methods from ours to control the insertion index are discussed in [31] and [23], where the insertion index is determined based on some scoring functions. It might be one of our future works to investigate whether their techniques fit in our algorithm.

We denote the average and the ratio by m_{in} and r_{in}, respectively, and define them as in Definition 5.

Definition 5. *The average of the index of insertion m_{in} is the current mean of the indices where ch_{in}s have been inserted. The ratio r_{in} is the current ratio of the number of times of insertion at the index j such that $m_{in} \leq j$ to the one at the index j such that $j < m_{in}$. When the denominator is* 0*, then r_{in} is defined to be* 0*.*

After a fixed number (e.g., 100) of restart runs are performed, if r_{in} is larger than a threshold r_{th}, we only insert ch'_{in}, which is an alternative of ch_{in}, at the index j such that $j < m_{in}$. The reason that a fixed number of restart runs need to be already performed is that the value of r_{in} is quite unstable in the initial stage of execution of our algorithm, but it becomes stable gradually. ch'_{in} is defined in the same way as in Definition 3 except that n at the equation of l in Definition 3 are substituted with $m_{in} - 1$. We can expect that this relieves the stagnation because if the insertion concentrates in the relative back

indices, r_{in} gets larger than r_{th}, and hence the insertion jumps in the relative front indices under the above constraint.

We noticed in one of our preliminary experiments, where we used our incomplete algorithm into which all of the ideas discussed in this paper are not incorporated, that a value of around 2.5 might be appropriate for r_{th}; in the 120-dimensional SVP challenge lattice, our incomplete algorithm found an optimal solution in 2–5 months for each value of r_{th} which is set to 2.0, 2.25, 2.5, 2.75, and 3.25, while it could not find an optimal solution during the same period for each value of r_{th} which is set to 1.5, 1.75, 3.0, and 3.5. This is, however, a limited experiment from which we can not obtain an optimal value for r_{th} with confidence. It might be also one of our future works to derive the theoretical value of r_{th}, but, in this paper, we attempt to determine the value of r_{th} flexibly by using the technique described in the following section.

5 Parameter Selection Based on Evolution Velocity of Lattice Basis

5.1 Problem of Parameter Selection

In Fukase's paper ([13]), Ding's paper ([9]) and Luan's paper ([21]) are referenced for determining the value of each genetic parameter, and detailed experiments for determining optimal values of parameters were not designed and conducted. The performance of GAs is greatly affected by parameter selection. We refer to e.g. [6] for the details. Therefore, there seems to be room for improving the performance of Fukase's algorithm in terms of parameter selection.

However, it is not necessarily an easy task to design and conduct experiments for determining optimal values of parameters. First, the running time measurements take too much time especially in high dimensions because the running time of GA changes significantly at every trial and hence the data should be averaged as stated in [13]. Secondly, there is no guarantee that the available data in already explored dimensions can be directly applied to unexplored dimensions. Thirdly, there is also no guarantee that, during the whole execution of GA, a fixed parameter set can maintain its effect despite the fact that the lattice basis changes at every restart. Therefore, in the following subsections, we address the problem discussed in this subsection, not by an experimental-based approach, but by another approach from a different angle.

Similar discussions to the above have occurred in the literature, especially in the context of self-adaptive genetic algorithms (SAGAs). In one particularly relevant study, Kivijärvi et al. [18] proposed a SAGA, where they adopted an evolutional approach to optimize their parameter values, for the clustering problem. In this paper, we also adopt an evolutional approach to addressing the problem of parameter selection not for the clustering problem but for the lattice problem. As shown in the next subsection, our evolutional approach has a marked difference compared to Kivijärvi's one. In Kivijärvi's SAGA, their genetic optimization for the parameter values is implemented simply as a random selection or an addition of a random number; for example, the mutation of their "crossover method" parameter corresponds to a random selection of one of their six crossover methods. On the other hand, in our GA, in addition to the search for an optimal solution, the parameters optimization in itself is also a GA, and this enables us to minutely refine our parameters.

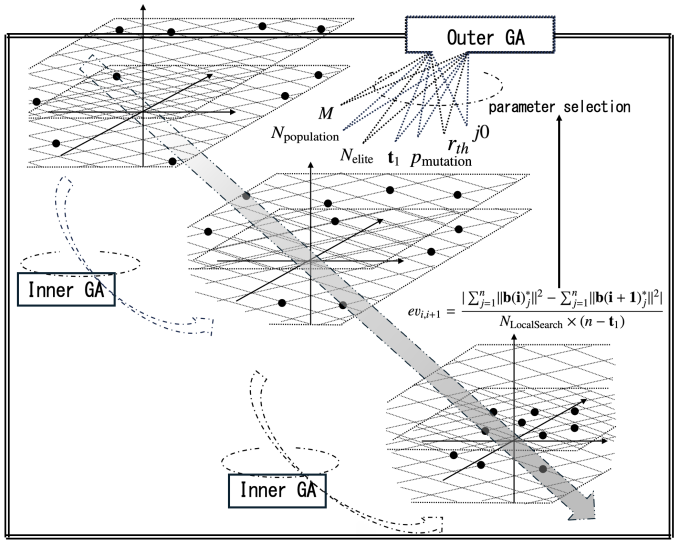

Fig. 7: The overview of our evolutional approach

5.2 Evolutional Approach to Parameter Selection

The key point of our evolutional approach is that, based on a genetic strategy, we optimize the parameters which are used in GA, and hence here our algorithm contains a double structure of GA. The inner structure corresponds to an improved version of Fukase's algorithm, where the improvement points discussed and addressed in Sections 3 and 4 are reflected. On the other hand, the outer structure corresponds to the parameters optimization, which in itself is also a GA. In the following, we call the inner and outer structures the inner and outer GAs, respectively. The overview of our approach is described in Figure 7.

In our extension of our algorithm, the outer GA does not change the structure of the inner GA at all. In fact, from the perspective of the inner GA, the only change brought by our extension is that each value of the parameters will be updated periodically. On the other hand, the outer GA uses a criterion for evaluation of chromosomes, which is designed to quantitatively measure the evolution of the lattice basis that is repeatedly updated by the inner GA. We call this criterion the evolution velocity and define it as in Definition 6. In the following section, we validate Definition 6 based on a discussion on time complexity after we describe our whole algorithm.

Definition 6. *For some integer $i \geq 0$, let B_i and B_{i+1} be the lattice bases used between the i-th and the $(i+1)$-th restarts and between the $(i+1)$-th and the $(i+2)$-th restarts, respectively, and let $[\mathbf{b(i)}_1^*, \ldots, \mathbf{b(i)}_n^*]$ and $[\mathbf{b(i+1)}_1^*, \ldots, \mathbf{b(i+1)}_n^*]$ be the Gram-Schmidt orthogonalized vectors corresponding to B_i and B_{i+1}, respectively; the 0-th restart corresponds to the initial reduction. The evolution velocity between the i-th and the $(i+1)$-th restarts, denoted by $ev_{i,i+1}$, is the ratio of the absolute difference between the two G-S*

sums to the total cost factor of local search between that time period, such that

$$ev_{i,i+1} = \frac{|\sum_{j=1}^{n} \|\mathbf{b}(\mathbf{i})_j^*\|^2 - \sum_{j=1}^{n} \|\mathbf{b}(\mathbf{i}+\mathbf{1})_j^*\|^2|}{N_{\mathrm{LocalSearch}} \times (n - \mathbf{t}_1)} \tag{1}$$

with the total number of chromosomes evaluated in local search $N_{\mathrm{LocalSearch}}$ *and the factor which determines the search space size* $(n - \mathbf{t}_1)$*. Here, the total cost factor of local search is represented with the product of* $N_{\mathrm{LocalSearch}}$ *and* $n - \mathbf{t}_1$*, where* $\mathbf{t}_1$ *is as in Definition 2, and is assumed to be always positive.*

In Definition 6, we take the absolute difference between the two G-S sums, which means that we also regard the increase of the G-S sum between some two restarts as an evolution. This might seem somewhat peculiar, considering that we aim to decrease the G-S sum as much as possible. As explained in Section 4.1, however, in the process of decreasing the G-S sum, if $\|\mathbf{b}_j^*\|^2$ is decreased for some j, $\|\mathbf{b}_k^*\|^2$ is inevitably increased for some ks such that $k \neq j$, and hence a temporary increase of the G-S sum can happen. Therefore, we define the evolution velocity so that it reflects both the increase and the decrease of the G-S sum, taking the absolute difference between the two G-S sums.

We assume that the denominator of Equation (1) is always positive. To be more precise, we have to design the lengths and the structures of the chromosomes which are used in the outer GA so that this condition is satisfied. In the following subsection, we present an example of such a design.

While the inner GA optimizes the squared Euclidean norm of a lattice vector, the outer GA optimizes the above evolution velocity. The former objective function evaluates a chromosome corresponding to a lattice vector in the form of NNR. On the other hand, the latter objective function evaluates each of chromosomes corresponding to the parameters which are used in the inner GA. In the following two subsections, we explain how to convert each of the parameters into the corresponding chromosome and how to evolve those chromosomes with reference to the evolution velocity of the lattice basis used in the inner GA.

5.3 Conversion of Parameters into Chromosomes

The parameters used in the inner GA are presented in Table 1. Except for the threshold of r_{in} and the maximum index of the last nonzero element of NNR, denoted by r_{th} and $j0$, respectively, the reader can refer to [13] for detailed descriptions in context where it is described what each parameter is used for and what values each parameter is likely to take. For the reader's convenience, however, we briefly recall some properties related to the two, somewhat obscure, parameters $\mathbf{t}$ and M in the following two paragraphs. With regards to $j0$, we explain it in this section later.

The parameter of NNR $\mathbf{t}$ is as in Definition 2 and is used to restrict each element of NNR to a small unsigned integer value. Except for $\mathbf{t}_1$, $\mathbf{t}$ is only used in the initialization step and can be roughly determined as explained in [13]. On the other hand, $\mathbf{t}_1$ represents the index range where each element of NNR can take a non-zero unsigned integer value by definition and hence determines the search space size of the local search procedure. Therefore, we attempt to optimize the value of $\mathbf{t}_1$ alone.

Table 1: Parameters used in the inner GA

Parameter	Notation
restart strategy parameter	M
parent population size	$N_{\text{population}}$
number of elites	N_{elite}
parameter of NNR	$\mathbf{t}$
mutation rate	p_{mutation}
threshold of r_{in}	r_{th}
maximum index of last nonzero element of NNR	$j0$

The restart strategy was introduced originally in [21], and the restart strategy parameter M is a fixed maximum number of successive generations with no update of the best chromosome. Since Fukase's algorithm includes a local search procedure, the update of the best chromosome is supposed to be accelerated, and hence, in [13], they set M to a much smaller number compared with that used in [21].

We convert each of the above parameters into a chromosome (a string) in a manner where our encoding is to convert a decimal number to the corresponding binary number for each parameter value; these binary strings are directly used as chromosomes. However, two things to note are the length and the structure of the chromosome for each parameter. For example, if the parent population size $N_{\text{population}}$ is set to an extraordinarily large number, it might cause an endless calculation during one generation. This is possible because a very high-order bit might be randomly flipped to 1 by mutation when the chromosome length is unlimited. Another case in point is that if the structure of the chromosome corresponding to the mutation rate p_{mutation} is not adjusted to avoid p_{mutation} being a number relatively close to 1, at some stage all genes of each chromosome used in the inner GA will be almost destroyed by mutation and the evolution process in the inner GA will consequently collapse. With these reasons in mind, we design the length and the structure of the chromosome for each parameter as in Figure 8.

In Figure 8, an example of the chromosome design for the input value of each parameter is presented for the 120-dimensional SVP challenge lattices. Here, it has to be emphasized that the chromosome design that is described in Figure 8 can be applied to any dimension without any customization; it is the input values that might need to be customized, and even if that is the case, it is not necessary to minutely determine them because the outer GA automatically adjusts them later.

Since Figure 8 alone can not explain all of the chromosome design, in the following, we offer some comments as a way of providing additional explanation of the chromosome design for each parameter. The first thing to state is that the chromosome which corresponds to each parameter is the first individual in the initial generation and how to generate the chromosomes of other individuals is explained in the following subsection. For the sake of convenience, we denote the binary length of the input value for each pa-

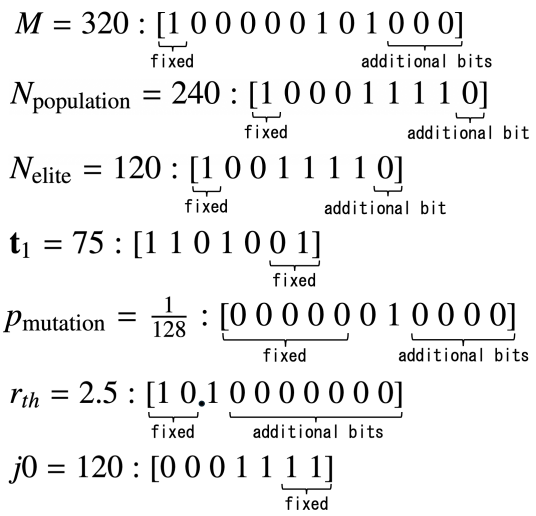

Fig. 8: An example of the chromosome design

rameter by *len*, where the symbol in subscript is used to identify the corresponding parameter name.

M The chromosome length is set to $len_M + 3$, where the additional 3 bits ensure that this parameter can take about 1–16 times larger values than the input one. Independently of the input value, the least significant bit is fixed to 1 (i.e., is not affected by crossover and mutation) to ensure that there is at least one generation between two successive restarts; if all bits are equally affected by crossover and mutation, all 0-bits can happen with some probability.

$N_{\text{population}}$ The chromosome length is set to $len_{N_{\text{population}}} + 1$, where the additional one bit ensures that this parameter can take about 1–4 times larger values than the input one. Independently of the input value, the least significant bit is fixed to 1 to ensure that there is at least one individual in each generation between two successive restarts; otherwise, there can not be any evolution between that time period.

N_{elite} The chromosome length is set to $len_{N_{\text{elite}}} + 1$, where the additional one bit ensures that this parameter can take about 1–4 times larger values than the input one. Independently of the input value, the least significant bit is fixed to 1 to ensure that at least one elite individual can always survive in the process of forming a new population between two successive restarts. If N_{elite} is set to a value larger than $N_{\text{population}}$, N_{elite} is reset to $N_{\text{population}}$; the number of elite individuals can not exceed the population size.

$\mathbf{t}_1$ The chromosome length is set to len_{t_1}. The most significant two bits are fixed as those of the input value to ensure that this parameter does not take a value much smaller or much larger than the input one.

p_{mutation} The input value (the decimal fraction) is converted to the binary fraction equivalent, and the chromosome length is set to $len_{p_{\text{mutation}}} + 4$, where $len_{p_{\text{mutation}}}$ ignores the

0 in the integer part and where the additional 4 bits ensure that the detailed adjustment can be done in the range of possible values. The input value is chosen so that the denominator is close to the lattice dimension n under the constraint that the denominator is divisible by a power of 2. The bits before the bit next to the one equal to 1 on the left are fixed to 0s to ensure that this parameter does not take a value much larger than the input one. If p_{mutation} is set to 0, the least significant bit is switched to 1 to avoid the absence of mutation.

r_{th} The input value (the decimal fraction) is converted to the binary fraction equivalent, and the chromosome length is set to $len_{r_{th}}+7$, where the additional 7 bits ensure that the detailed adjustment can be done in the range of possible values. The input value is chosen so that $len_{r_{th}}$ is relatively short. The dot at the chromosome corresponding to r_{th} in Figure 8 represents the decimal point. In this example, the bits before the dot (i.e., the integer part) are fixed as those of the input value to ensure that this parameter does not take a value much smaller or much larger than the input one.

$j0$ The chromosome length is set to len_{j0}. The most significant two bits are fixed as those of the input value to ensure that this parameter does not take a value much smaller than the input one. If $j0$ is set to a value larger than the lattice dimension n, $j0$ is reset to n. In addition, if $j0$ is set to a value smaller than $n-7$, $j0$ is reset to $n-7$ so that $j0$ is kept close to n. This value, $n-7$, is selected based on one of our preliminary experiments; in the experiment, we tested the performance with our complete algorithm in the range of values of $j0$ from $n-10$ to n.

We note that the chromosome design shown above is an example, although the chromosome length and the range where each bit is fixed are examined in detail for each parameter based on our experimental observations. It is one of our future works to seek a better chromosome design.

5.4 Evolution of Parameters Based on Evolution Velocity

Here, we explain how the outer GA evolves the chromosomes corresponding to each parameter with reference to the evolution velocity. The first thing to state is that the optimal solution which the outer GA attempts to obtain is a set of values for the parameters which enables the inner GA to identify a very short lattice vector satisfying the success condition for the SVP challenge, although it is the evolution velocity that the outer GA explicitly optimizes. For each parameter, the outer GA approaches towards its optimal solution as follows.

Step 1 The input value is converted to the binary string according to the chromosome design that is described in Figure 8, and this string is the chromosome of the first individual in the initial generation. The population size is set to $2len$ for the chromosome length len, and the chromosomes of other individuals correspond to $2len-1$ strings each of which is obtained by converting a random value close to the input one to the binary string.

Step 2 The outer GA is started with the above initial generation.

Step 3 The chromosomes take turns being evaluated. Each turn corresponds to the execution of the inner GA between update reductions, and this execution corresponds

to the fitness evaluation. Each chromosome gets linked to the value of the evolution velocity available at the time when the evolution velocity is updated after the restart at the end of the corresponding execution of the inner GA. At the beginning of the fitness evaluation, each chromosome is converted from the binary string to its decimal equivalent and then serves the inner GA as one of the parameters used there. The whole algorithm where the inner and outer GAs are run terminates if the inner GA identifies a very short lattice vector satisfying the success condition for the SVP challenge, and the set of values for the parameters for that point in time is the optimal solution obtained by the outer GA.

Step 4 To form a new population, a mix ranking-based and random selection with elitist strategy is used; a fixed number of the best chromosomes in the last generation of population form a new population along with a fixed number of the ones produced by crossover and mutation. In random selection, first, mating parents from the current population are randomly selected, secondly, crossover between each parent pair is done, and thirdly, random mutation of chromosomes is done. The outer GA then returns to step 3.

The cycle in which a new population is formed differs for each parameter because the population size is twice the chromosome length, which generally differs for each parameter. In ranking-based selection with elitist strategy, with a larger value of the evolution velocity, a chromosome gets a higher rank. On the other hand, newborn offsprings generated through the process of random selection, unlike their parents' generation, can not be ranked at that point in time because they can not be evaluated in terms of the evolution velocity before the following executions of the inner GA between update reductions. Thus, we maintain both high-rank individuals and the diversity in the population by the above strategy.

The whole algorithm does not suffer from any increase in the runtime cost during the fitness evaluation of the outer GA because, as explained above, it corresponds to the execution of the inner GA. In addition, in the outer GA, the runtime cost for other genetic operations, where there are only the same number of copies of chromosomes as the population size to dominate the runtime, is negligible compared to that for the fitness evaluation. Therefore, the outer GA can serve the whole algorithm almost without the runtime cost.

5.5 Extensibility of Our Algorithm

When our algorithm is improved or extended, it can probably adopt some new parameters without any design changes of itself and without any manual parameter tuning. In this subsection, we explain this extensibility of our algorithm. Here, we attempt to introduce our version of "a few dimensions for free", the original version of which is a technique used in lattice sieving [2, **?**]. Their technique uses the orthogonal complement of the linear space spanned by a subset of the Gram-Schmidt vectors of a lattice basis, and hence its dimension is lower than the lattice dimension n. Sieving finds short vectors in the orthogonal complement, and then those vectors are lifted to the corresponding vectors in higher dimensions by using Babai Lifting [4].

Their original version of "a few dimensions for free" and ours are completely different except that we also search short vectors in dimensions lower than the lattice dimension n. Our technique takes advantage of the property of NNR. More specifically, we fix a few elements of NNR from the rear to 0. By doing this, we can narrow the search space of the local search procedure. Let $j0$ be the maximum index of the last nonzero element of NNR, and then this is identical to searching in the lattice of dimension $j0$. In this sense, our technique is a kind of "a few dimensions for free". The main point is how many dimensions we can make for free and that we can leave this issue to our outer GA; our outer GA will determine the optimal value of $j0$ flexibly. The only three things to be done are to design the length and the structure of the chromosome for $j0$ as in Figure 8, to replace the lattice dimension n with $j0$ in the local search procedure and at the denominator of Equation (1) in Definition 6, and to fix each of the genes of which the indices are larger than $j0$ to 0 in the initialization of the chromosomes linked to lattice vectors.

6 Our Whole Algorithm

In this section, we obtain a double-structured GA, in which we integrate the two improvement points mentioned in Sections 3 and 4 and into which we incorporate the outer GA introduced in Section 5.

6.1 Description of Our Algorithm

Our algorithm (a double-structured GA) is described in detail in Algorithm 2. We present the pseudocode as detailed as possible in a C-like syntax to ensure the reader can implement the algorithm illustrated in this paper, although the total code length seems to be quite long as a result of that. In fact, our pseudocode seems to be closer to a real programming language and to be more detailed, compared to the ones presented in [9] and [21].

In the following, for the better readability of the pseudocode, we hypothetically separate several subroutines from the main routine, although, of course, the reader does not necessarily need to follow the following subroutinization in the actual implementation. In our pseudocode, we hence do not restrict the scope of variables, and each variable is assumed to be accessed from any scope without global declaration; otherwise, our pseudocode would have been longer and more complicated to understand.

Our algorithm uses several subroutines which are the same as those as in Fukase's algorithm [13]; it uses *FitnessEvaluation*, *Crossover*, *Mutation*, and *Decode*, and in addition uses three subroutines for lattice operations, *BKZ*, *ComputeGS*, and S^2*LLL* in the same way as in Fukase's algorithm. Except for S^2*LLL*, all above subroutines which are used in Algorithm 2 are exactly the same as those as in Fukase's algorithm, but for the self-containedness of this paper we present the detailed descriptions of them as they are in [13] (Algorithms 5, 7, 8, and 10). The reader can refer to [13] for full details. The only difference between Fukase's implementation of S^2*LLL* and ours is that the latter does not include LLL reduction.

We use our own five genetic subroutines aside from the above subroutines: *Initialization*, *InitializationOuter*, *FitnessEvaluationEX*, *LocalSearch*, and *OuterGA*.

Our *Initialization* is described in detail in Algorithm 3. In *Initialization*, as stated in Section 5.5, each of genes of which the indices are larger than $j0$ is fixed to 0. Other than that, *Initialization* is essentially the same thing as Fukase's initialization subroutine, but the former, unlike the latter, does not define and does not allocate memory to variables of population in this step. In our algorithm, as explained later in this subsection, we define and allocate memory to them at another place while being careful to reserve the required size of memory for the possible maximum size of population.

Our outer GA is separated into *InitializationOuter* and *OuterGA*; *InitializationOuter* initializes the chromosomes corresponding to each parameter, and *OuterGA* conducts a series of genetic operations for those chromosomes.

Our *FitnessEvaluationEX* is described in detail in Algorithm 6. It has three distinct differences against Fukase's *FitnessEvaluation* in [13]; first, it computes and updates ch_{in} defined in Definition 3 in addition to computing the squared Euclidean norm of a lattice vector, secondly, it includes the method of controlling the insertion index using the two measures m_{in} and r_{in} defined in Definition 5 and thirdly, it finds and updates the chromosome which corresponds to the current shortest lattice vector.

Since our elitist strategy keeps not the current shortest lattice vector but the current best chromosomes that correspond to a fixed number of NNRs, when the new current shortest lattice vector is found, it might be lost unless it is inserted in the current lattice basis for a restart run. Here, we recall that these chromosomes themselves do not directly include information about the lattice basis or the corresponding lattice vector and that they only include the value of the corresponding norm. To avoid this loss, we always keep the current shortest lattice vector aside from the current best chromosomes in our elitist strategy.

The local search technique is used also in Fukase's algorithm in [13], but there is a distinct difference between Fukase's local search subroutine and our *LocalSearch*. It is that the latter uses *FitnessEvaluationEX* besides *FitnessEvaluation*; other than that difference, there are no differences between them. It is described in detail in Algorithm 9.

Here, we offer some comments as a way of providing additional explanation of some of our codes.

from Line 2 to Line 10 of Algorithm 2 For each variable of population, the required size of memory is reserved for the possible maximum size of population. First, the possible maximum size of population $N_{\text{population_max}}$ is calculated according to the chromosome design that is described for $N_{\text{population}}$ in Figure 8. Here, $\text{BN}(N_{\text{population}})$ returns the binary length of $N_{\text{population}}$. S_{parent} is a structure array each element of which consists of a lattice vector in NNR form and its norm. S_{parent} stores all individuals in the parent generation. Secondly, for the offspring generation in the same way as for the parent generation, $N_{\text{offspring_max}}$ and $S_{\text{offspring}}$ are calculated and allocated in memory, respectively.

from Line 11 to Line 16 of Algorithm 2 m_{in}, several variables related to m_{in}, and r_{in} are initialized.

from Line 17 to Line 19 of Algorithm 2 $\mathbf{z}_{in}$ and $\mathbf{z}'_{in}$ are the chromosomes of the current and the previous candidates for achieving the possible maximum extent of decrease, respectively. $\mathbf{z}_{sh}$ is the chromosome of the current shortest lattice vector. An array of int values with the size equal to n is assigned to each chromosome.

from Line 32 to Line 35 of Algorithm 2 d is the possible maximum extent of decrease in $\|\mathbf{b}^*_j\|^2$ between update reductions and is initialized. i_{in} is the index of insertion of ch_{in} and is initialized. *is_candidate* is a flag variable which corresponds to whether the current chromosome is a candidate for achieving the possible maximum extent of decrease and is initialized. *cnt_localsearch* is a counter variable used to keep track of the number of times the local search procedure has been conducted and is initialized.

from Line 70 to Line 82 of Algorithm 2 Here, the duplication of chromosomes is eliminated. The current value of $kk - k$ equals the current number of the chromosomes discarded due to duplication.

Line 84 of Algorithm 2 A restart run is called with the current B if $d = 0$ (i.e., the extent of decrease equals zero), which means any ch_{in} or ch'_{in} has not been found.

from Line 85 to Line 94 of Algorithm 2 m_{in} and r_{in} are computed. r_{in} remains initialized with the default value 0 while the denominator equals zero.

from Line 95 to Line 98 of Algorithm 2 Here, the update reduction is conducted. $\mathbf{z}_{in}$ (ch_{in}) is decoded into the corresponding lattice vector $\mathbf{v}_{in}$ in $L(B)$, where the notation between ch_{in} and ch'_{in} is unified into the former ch_{in} ($\mathbf{z}_{in}$). *BKZ* then reduces a generating system G consisting of B and $\mathbf{v}_{in}$ and removes a zero-vector, which is produced due to the fact that G is a linearly dependent system.

Line 100 of Algorithm 2 the G-S sum before the update reduction is preserved to be used in the computation of the evolution velocity *ev*.

definition of two structure arrays in Algorithm 4 These two structure arrays for other parameters are also defined in the same way here except that the notation M added in subscript to the names of the two structure arrays is substituted with the notation of each parameter, and those definitions are omitted for avoiding repetition and lengthy descriptions. In the following codes, if possible, we omit the same description patterns in the same way as here.

from Line 2 of Algorithm 4 Here, chromosomes corresponding to the parameters which are used in the inner GA are initialized according to the chromosome design that is described for each parameter in Figure 8. For each parameter, the population size of the chromosomes is set to twice the chromosome length, and an array of parameter structures with the size equal to the population size is assigned to the parents generation, where each parameter structure contains a binary string $\mathbf{z}$ and a double variable *ev*.

from Line 3 to Line 10 of Algorithm 4 $\mathrm{BS}(M, len_M)$ returns the binary string of M with the length equal to the given length len_M, where some highest order bits are set to 0s if $len_M > \mathrm{BN}(M)$. $\mathrm{Copy}(S_{\mathrm{parent}_M}[1].\mathbf{z}, \mathrm{BS}(M, len_M))$ copies the second argument $\mathrm{BS}(M, len_M)$ to the first one $S_{\mathrm{parent}_M}[1].\mathbf{z}$, and Rand() returns a random number uniformly distributed in the range $[0, 1)$, including only the left end point. Here, the chromosome of the first individual in the generation is set to $\mathrm{BS}(M, len_M)$, and the chromosomes of other individuals are set to $\mathrm{BS}(r, len_M)$ with an integer r which behaves as a random integer close to M.

Line 14 of Algorithm 4 Each element of $S_{\text{offspring_}M}$ is nothing more than an int array of length len_M; at that point in time, each element of $S_{\text{offspring_}M}$ is not linked to the corresponding evolution velocity ev, the value of which can not be given before the fitness evaluation of the chromosome which corresponds to the element.

from Line 60 to Line 78 of Algorithm 4 $\text{FBN}(r_{th})$ returns the binary length of r_{th}. $\text{FBS}(r_{th}, len_{r_{th}})$ returns the binary string of r_{th} with the length equal to the given length $len_{r_{th}}$, where some lowest order bits are set to 0s if $len_{r_{th}} > \text{FBN}(r_{th})$. The chromosome of the first individual in the generation is set to $\text{FBS}(r_{th}, len_{r_{th}})$, and the chromosomes of other individuals are set to $\text{FBS}(r, len_{r_{th}})$ with a floating-point number r which behaves as a random floating-point number close to r_{th}.

Line 23 of Algorithm 6 This if-statement is confusing, but in short, in the context of it, if restart runs have been already performed more than 100 times and $r_{in} > r_{th}$, then ch'_{in} is searched instead of ch_{in}.

from Line 24 to Line 31 of Algorithm 6 This if-statement is run for preventing the local basis $(\mathbf{b}_1, \ldots, \mathbf{b}_{i_{in}-1})$ from being influenced by the insertion of ch_{in} if the current candidate turns out not to satisfy the requirement in Definition 3. In this if-statement, the current candidate is discarded, and the previous candidate is recalled.

from Line 32 to Line 44 of Algorithm 6 This if-statement is run if the current chromosome turns out to be a candidate for achieving the possible maximum extent of decrease. Here, the previous candidate is kept temporarily in case the current chromosome turns out not to be a candidate later.

from Line 45 to Line 50 of Algorithm 6 If a chromosome which corresponds to a shorter lattice vector than ever before is found, the current shortest lattice vector is updated.

Line 1 of Algorithm 11 The evolution velocity ev is computed according to Definition 6.

from Line 2 to Line 38 of Algorithm 11 A series of genetic operations for the chromosomes corresponding to M is performed. Since the outer GA evaluates the chromosomes in turn at every restart run, we move the index of $S_{\text{parent_}M}$ forward taking the remainder of $cnt_restart - 1$ when divided by $N_{\text{population_}M}$ and adding 1 to it, and hence a generation will have been finished if the index equals $N_{\text{population_}M}$. In mutation step, the for-loop (Line 26) is restricted so that the gene which is fixed to 1 in Figure 8 is never switched.

6.2 Validation of Definition of Evolution Velocity

Here, we validate Definition 6 in Section 5.2 based on a discussion on time complexity. In the following, $\Omega(\cdot)$ and $O(\cdot)$ denote the lower and upper bounds of the asymptotic complexity, respectively. $\Theta(\cdot)$ denotes both the lower and upper bounds of the asymptotic complexity. First, we prove the following proposition. After that, we explain the validity of the denominator of Equation (1) in Definition 6 by representing the time complexity given in the proposition as another form.

Proposition 1. *Assuming $M = \Theta(n)$ and the time complexity of the number of repetition of the repeat–until in the lines 47–54 of Algorithm 2 is $\Theta(1)$, the time complexity*

between the i-th and the $(i + 1)$*-th restarts for some integer* $i \geq 0$ *is* $\Omega(n^5)$ *with lattice dimension n.*

Proof. The time complexities of both Crossover and Mutation, the former and latter of which contain two single for-loops and a single for-loop, respectively, are $\Theta(n)$; the time complexity of each of these for-loops is $\Theta(n)$.

On the other hand, LocalSearch contains a single while-loop, of which the time complexity of the number of repetition is $\Omega(1)$ and where FitnessEvaluationEX is executed in a single for-loop. The time complexities of FitnessEvaluationEX and the number of repetition of this for-loop are $\Theta(n^2)$ and $\Theta(n)$, respectively; FitnessEvaluationEX contains a double for-loop, of which the time complexity is $\Theta(n^2)$. Therefore, the time complexity of LocalSearch is $\Omega(n^3)$.

In the repeat–until in the lines 47–54 of Algorithm 2, the time complexities of both Crossover and Mutation are negligible compared to that of LocalSearch, and based on the assumption and with $N_{\text{population}} = \Theta(n)$, the time complexity of the while-loop at line 45 of Algorithm 2, of which the time complexity of the number of repetition is $\Omega(M)$, is $\Omega(n^5)$. Here, among other steps in this while-loop, Sort has the maximum time complexity $O(n^2)$ in the worst case, and this is negligible compared to that of the for-loop at line 46 of Algorithm 2.

Outside of the above while-loop and in the range between two successive restarts, the for-loop at line 38 of Algorithm 2 has the maximum time complexity $\Theta(n^3)$; this for-loop contains FitnessEvaluation, of which the time complexity is $\Theta(n^2)$. This is negligible compared to that of the above while-loop. Therefore, the total complexity between two successive restarts is $\Omega(n^5)$.

According to the above proof, the total complexity between two successive restarts is $\Omega(n^5)$, and this complexity can be more strictly evaluated and be represented also as $N_{\text{LocalSearch}} \times (n - \mathbf{t}_1) \times \Omega(n^2)$, where $N_{\text{LocalSearch}}$ and $\mathbf{t}_1$ are as in Definitions 6 and 2, respectively, by viewing our pseudocode in more detail. Therefore, with the fixed lattice dimension n, the total complexity between two successive restarts can be considered to be approximately proportional to the product of $N_{\text{LocalSearch}}$ and $n - \mathbf{t}_1$, and this explains the validity of the denominator of the evolution velocity, which is as in Equation (1) in Definition 6 and which is used to measure the runtime cost between that time period; in our complete algorithm, as explained in Section 5.5, the lattice dimension n is replaced with $j0$ at the denominator of the evolution velocity, and this does not influence the above discussion since it can be assumed that $j0 = \Theta(n)$ according to our chromosome design.

6.3 Performance Evaluation

Using our complete algorithm, we ran 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice. Table 2 shows the result of these trials. The column headed "running time (day)" shows, for each trial, the running time (day) spent for finding an optimal solution; we terminated the execution for each trial if the running time was over 5 months. The column headed "relative length" shows, for each trial, the ratio of the Euclidean norm of the shortest lattice vector ever found until the execution was terminated to Gaussian Heuristic $GH(L)$.

Table 2: The result of 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice.

trial no.	running time (day)	relative length
1	not terminated in 5 months	1.06238
2	126	1.013
3	24	1.04037
4	not terminated in 5 months	1.06014
5	not terminated in 5 months	1.06157
6	not terminated in 5 months	1.06393
7	45	0.99535
8	16	1.03042
9	53	1.03564
10	not terminated in 5 months	1.06393

To compare the performance of our algorithm with that of Fukase's algorithm [13], using Fukase's algorithm, we ran the same calculations as in the case of our algorithm. Table 3 show the result of these calculations and can be read in the same way as in Table 2. Here, we ran the above calculations in both cases under the same conditions over execution time and reduction methods for restarts (BKZ and S^2LLL) including their parameter values to offer a fair comparison. We can see from Tables 2 and 3 that our algorithm succeeded in finding an optimal solution within several months in 5 out of the 10 trials, while, using Fukase's algorithm, an optimal solution was not found in 5 months in all of the 10 calculations. We can also see that, in Table 2 even in the trials where an optimal solution was not found, the lerative lengths closely approach the success condition 1.05 for the SVP challenge, while those in Table 3 are much larger than 1.05. Considering these comparisons, we can be confident that our algorithm realizes a significant performance gain.

In the calculations of Table 2, the outer GA provided two beneficial effects; (i) it realized flexible parameter selection, and (ii) it accelerated our incomplete algorithm which only includes the techniques that are shown in sections up to and including Section 4.2. First, concerning parameter selection, in all of the 10 trials, we used each value shown in Figure 8 as the input one for each parameter. For example, in one out of the 10 trials where our algorithm found an optimal solution within a month, these input values evolved, finally, into the following values: $M = 2419$, $N_{\text{population}} = 329$, $N_{\text{elite}} = 41$, $\mathbf{t}_1 = 88$, $p_{\text{mutation}} = 0.00292969$, $r_{th} = 2.98047$, and $j0 = 115$. The significance of our outer GA lies in the fact that, as in this example, it enables delicate parameter tuning during the execution of our algorithm.

Secondly, concerning acceleration effect, as we can see in Table 2, in 2 out of the 10 trials, our algorithm found an optimal solution within a month, while our incomplete algorithm into which the outer GA is not incorporated never succeeded during the same period. Also in one of our preliminary experiments which was conducted using our complete algorithm to examine better chromosome design for each parameter, there is

Table 3: The result of 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice.

trial no.	running time (day)	relative length
1	not terminated in 5 months	1.1303
2	not terminated in 5 months	1.12578
3	not terminated in 5 months	1.10314
4	not terminated in 5 months	1.14683
5	not terminated in 5 months	1.13112
6	not terminated in 5 months	1.13807
7	not terminated in 5 months	1.11378
8	not terminated in 5 months	1.12633
9	not terminated in 5 months	1.12125
10	not terminated in 5 months	1.12562

Table 4: The result of 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice.

case	the average relative lengths	standard deviations
normal	1.1145	0.01898
random	1.1462	0.01658
reciprocal	1.1487	0.0169

a trial where an optimal solution was found within a month. Thus, we experimentally confirmed the effectiveness of our outer GA.

In addition, to make sure that the evolution velocity, which is the objective function of the outer GA, guides our algorithm towards populations containing chromosomes which correspond to shorter lattice vectors, we conducted an experiment of comparison of three cases. In our program, in one case we intentionally and always assigned a random number to the evolution velocity ev, and furthermore in the other case we intentionally and always took the reciprocal of the evolution velocity (i.e., $1/ev$); in the latter case, we assigned a large number (we chose 1.0e + 10) to the variable ev to avoid zero division when ev is zero. We compared the search quality in both cases with that in the normal case where ev is assigned as in Definition 6. If the evolution velocity has the ability to guide our algorithm towards the success of the search, the search quality is supposed to be high in the order of "normal", "random", and "reciprocal". We ran 100 trials of search in the 120-dimensional SVP challenge lattice in all of the three cases. In this experiment, we set the runtime to 1 week per trial. Table 4 shows the result of this experiment. We can see in Table 4 that, in both "random" and "reciprocal" cases, the search quality was degraded and that it is high in the order of "normal", "random", and "reciprocal" as we expected above. Thus, we confirmed that the evolution velocity guides our algorithm towards the success of the search.

Table 5: The result of 10 trials of a parallel version of our algorithm for the 120-dimensional SVP challenge lattice.

trial no.	running time (day)	relative length
1	11	1.03826
2	38	1.04318
3	57	1.04058
4	28	1.0
5	22	1.04938
6	4.8	1.04907
7	7.7	1.02867
8	not terminated in 5 months	1.0571
9	42	1.0
10	13	1.02126

7 Conclusion and Future Work

In this paper, we proposed the first double-structured and the fastest GA for solving the SVP to our knowledge and to date, respectively. Our algorithm consists of the inner and the outer GAs, is multiobjective (double objective), and uses two types of chromosomes. Our originality lies especially in this point. We experimentally confirmed that our outer GA, of which the objective function is our novel criterion "the evolution velocity", realized flexible parameter selection, which is also not included in the existing GAs. In our experiments, our algorithm could solve SVP in dimension 120 without using any parallelization technique. Our dimension 120 is higher than Ding's dimension 118, which is the previous highest record in GAs for solving the SVP.

Our algorithm is definitely able to be further accelerated and to proceed to higher-dimensions, subject to some efficient parallelization and performed using sufficiently large computational resources. In fact, we have implemented a parallel version of our algorithm simply by file sharing, and this test version over 8 CPU processes could solve SVP in dimension 120 in 4.8 days at its best; using this parallel version, we ran 10 trials of search for an optimal solution in the 120-dimensional SVP challenge lattice, and Table 5, which can be read in the same way as in Table 2, shows the result of these trials. By comparing Tables 2 and 5, we can see that this parallel version showed some acceleration effect. In our future work, based on this test version, we develop actual parallel versions to pursue further acceleration effects.

Making GAs one of the classes of fast algorithms is the aim of our future work, where we tackle the following two problems: (i) pursuing acceleration on a single thread on a single core (without using any parallelization technique) by further improvements of our algorithm itself and (ii) constructing parallel GAs which are derived from our algorithm and which achieve significant speedups by using plentiful techniques on parallel acceleration developed in the field of GAs.

Concerning the first point (i), it might be effective to investigate whether other structures fit in our algorithm. For example, one of the candidates for such structures is the

combination of our GA with differential evolution (DE) [30], which is a variant of evolutionary algorithms, including GAs; for the details of DE, we refer to [8]. More specifically, we plan to use DE to bundle up the parameters used in our GA into parameter vectors, which correspond to genomes in DE. This might make our GA able to be implemented in a smaller number of codes, although it is quite unclear at this time and should be clarified in our future work to what extent the performance gain can be achieved by this. Other structures such as self-adaptive DE (e.g., we refer to [1]) or double-structured DE might be also worth investigation.

Acknowlegements The author was supported by the Telecommunications Advancement Foundation (TAF).

References

1. H.A. Abbass, "The Self-Adaptive Pareto Differential Evolution Algorithm," Proceedings of IEEE Congress on Evolutionary Computation, vol.1, pp.831-836(2002).
2. M.R. Albrecht, L. Ducas, G. Herold, E. Kirshanova, E.W. Postlethwaite, M. Stevens, "The General Sieve Kernel and New Records in Lattice Reduction," Proceedings of EUROCRYPT 2019, vol.11477 of LNCS, pp.717–746(2019).
3. Y. Aono, P.Q. Nguyen, "Random Sampling Revisited: Lattice Enumeration with Discrete Pruning," Proceedings of EUROCRYPT 2017, vol.10211 of LNCS, pp.65-102(2017).
4. L. Babai, "On Lovász' Lattice Reduction and the Nearest Lattice Point Problem," Combinatorica, vol.6, pp.1-13(1986).
5. A. Becker, L. Ducas, N. Gama, T. Laarhoven, "New Directions in Nearest Neighbor Searching with Applications to Lattice Sieving," Proceedings of SODA 2016, pp.10–24(2016).
6. O. Boyabatli, I. Sabuncuoglu, "Parameter selection in genetic algorithms," Journal of Systemics, Cybernetics and Informatics, vol.4, no.2, pp.78-83(2004).
7. J. Buchmann, C. Ludwig, "Practical Lattice Basis Sampling Reduction," Proceedings of ANTS 2006, vol.4076 of LNCS, pp.222-237(2005).
8. S. Das, P.N. Suganthan, "Differential Evolution: A Survey of the State-of-the-Art," IEEE Transactions on Evolutionary Computation, vol.15, issue.1, pp.4-31(2011).
9. D. Ding, G. Zhu, X. Wang, "A Genetic Algorithm for Searching the Shortest Lattice Vector of SVP Challenge," Proceedings of GECCO 2015, pp.823–830(2015).
10. L. Ducas, "Shortest Vector from Lattice Sieving: A Few Dimensions for Free," Proceedings of EUROCRYPT 2018, vol.10820 of LNCS, pp.125–145(2018).
11. L. Ducas, M. Stevens, W. van Woerden, "Advanced Lattice Sieving on GPUs, with Tensor Cores," Proceedings of EUROCRYPT 2021, vol.12697 of LNCS, pp.249–279(2021).
12. M. Fukase, K. Kashiwabara, "An Accelerated Algorithm for Solving SVP Based on Statistical Analysis," Journal of Information Processing, vol.23, no.1, pp.67-80(2015).
13. M. Fukase, M. Kaminaga, "Improving Genetic Algorithm for Solving SVP: Focusing on Low Memory Consumption and High Reproducibility," Iran Journal of Computer Science, vol.5, issue.4, pp.359-372(2022).
14. N. Gama, P.Q. Nguyen, O. Regev, "Lattice Enumeration Using Extreme Pruning," Proceedings of EUROCRYPT 2010, vol.6110 of LNCS, pp.257-278(2010).
15. J.J. Grefenstette, "Optimization of Control Parameters for Genetic Algorithms," IEEE Transactions on Systems, Man, and Cybernetics, vol.16, no.1, pp.122-128(1986).
16. T. Harada, E. Alba, "Parallel Genetic Algorithms: A Useful Survey," ACM Computing Surveys (CSUR), vol.53, no.4, pp.1-39(2020).

17. J.H. Holland, “Adaptation in Natural and Artificial Systems,” The University of Michigan Press(1975).
18. J. Kivijärvi, P. Fränti, O. Nevalainen, “Self-Adaptive Genetic Algorithm for Clustering,” Journal of Heuristics, vol.9, no.2, pp.113-129(2003).
19. P.C. Kuo, M. Schneider, Ö. Dagdelen, J. Reichelt, J. Buchmann, C.M. Cheng, B.Y. Yang, “Extreme Enumeration on GPU and in Clouds,” Proceedings of CHES 2011, pp.176–191(2011).
20. A. K. Lenstra, H. W. Lenstra, L. Lovász, “Factoring Polynomials with Rational Coefficients,” Mathematiche Annalen, vol.261, pp.213-219(1982).
21. L. Luan, C. Gu, Y. Zheng, “A Genetic Algorithm with Restart Strategy for Solving Approximate Shortest Vector Problem,” Proceedings of ICACI 2020, pp.243–250(2020).
22. C. Ludwig, “Practical Lattice Basis Sampling Reduction,” PhD thesis, TU Darmstadt(2005).
23. E. W. Postlethwaite, “Topics in Lattice Sieving,” PhD thesis, Royal Holloway, University of London(2021).
24. J.D. Schaffer, “Multiple Objective Optimization with Vector Evaluated Genetic Algorithms,” Proceedings of the 1st International Conference on Genetic Algo- rithms, pp.93–100(1985).
25. M. Schneider, N. Gama, “SVP Challenge,” Available at http://www.latticechallenge.org/svp-challenge/
26. M. Schneider, “Computing Shortest Lattice Vectors on Special Hardware,” PhD thesis, TU Darmstadt, (2011).
27. C.P. Schnorr, M. Euchner, “Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems,” Math. Programming, vol.66, pp.181-199(1994).
28. C.P. Schnorr, “Lattice Reduction by Random Sampling and Birthday Methods,” Proceedings of STACS 2003, vol.2607 of LNCS, pp.145-156(2003).
29. V. Shoup, “Number Theory C++ Library(NTL) version 11.5.1,” Available at https://www.shoup.net/ntl/
30. R. Storn, K.V. Price, “Differential Evolution: A Simple and Efficient Adaptive Scheme for Global Optimization over Continuous Spaces.pdf,” ICSI, USA, Tech. Rep. TR-95-012, pp.1-12(1995).
31. T. Teruya, K. Kashiwabara, G. Hanaoka, “Fast Lattice Basis Reduction Suitable for Massive Parallelization and Its Application to the Shortest Vector Problem,” Proceedings of PKC 2018, vol.10769 of LNCS, pp.437–460(2018).
32. M. Yasuda, J. Yamaguchi, “A New Polynomial-Time Variant of LLL with Deep Insertions for Decreasing the Squared-Sum of Gram-Schmidt Lengths,” Designs, Codes and Cryptography, vol.87, no.11, pp.2489–2505(2019).

Algorithm 2 Our algorithm

Input:
B: a lattice basis $[\mathbf{b}_1, \ldots, \mathbf{b}_n]$
β: the BKZ parameter
M: the restart strategy parameter
$\mathbf{t}$: a parameter of NNR
r_{th}: the threshold of r_{in}
Output:
$\mathbf{v}$: a nonzero lattice vector such that $\|\mathbf{v}\| < 1.05 \cdot GH(L(B))$

BKZ(B, β)
$N_{\text{population}} := 2n$
$N_{\text{elite}} := n$
$N_{\text{population_max}} := \text{Pow}(2, \text{BN}(N_{\text{population}}) + 1) - 1$
$S_{\text{parent}} := \text{Malloc}(\text{lattice_vector}, N_{\text{population_max}})$
// assign an array of lattice_vector structures with the size equal to $N_{\text{population_max}}$ to S_{parent}, where each lattice_vector structure contains a lattice vector (a chromosome) $\mathbf{z}$ in NNR form and a double variable *norm*.
$N_{\text{elite_max}} := \text{Pow}(2, \text{BN}(N_{\text{elite}}) + 1) - 1$
$N_{\text{offspring_max}} := N_{\text{population_max}} + N_{\text{elite_max}}$
$S_{\text{offspring}} := \text{Malloc}(\text{lattice_vector}, N_{\text{offspring_max}})$
// assign an array of lattice_vector structures with the size equal to $N_{\text{offspring_max}}$ to $S_{\text{offspring}}$
$m_{in} := n$
$sum_i_{in} := 0$
$cnt_restart := 0$
$cnt_i_{in}_s := 0$
$cnt_i_{in}_l := 0$
$r_{in} := 0$
$\mathbf{z}_{in} := \text{Malloc}(\text{int}, n)$
$\mathbf{z}'_{in} := \text{Malloc}(\text{int}, n)$
$\mathbf{z}_{sh} := \text{Malloc}(\text{int}, n)$
$norm_sh := \|\mathbf{b}_1\|$ // initialization of the norm of the current shortest lattice vector
$j0 := n$ // the maximum index of the last nonzero element of NNR
InitializationOuter($M, \mathbf{t}[1], r_{th}, N_{\text{population}}, N_{\text{elite}}, j0$)
while TRUE **do**
    ComputeGS($B, \boldsymbol{\mu}, \mathbf{c}$)
    $gs_sum := 0$
    for $i := 1, \ldots, n$ **do**
        $gs_sum := gs_sum + \mathbf{c}[i]$ // compute G-S sum
    end for
    if $cnt_restart > 0$ **then**
        OuterGA($\mathbf{t}[1], j0$)
    end if
    $d := 0$
    $i_{in} := -1$
    $is_candidate := 0$
    $cnt_localsearch := 0$
    $N_{\text{offspring}} := N_{\text{population}} + N_{\text{elite}}$
    Initialization($\mathbf{t}, N_{\text{population}}$)
    for $k := 1, \ldots, N_{\text{population}}$ **do**
        $S_{\text{parent}}[k].norm := \text{FitnessEvaluation}(S_{\text{parent}}[k].\mathbf{z}, \boldsymbol{\mu}, \mathbf{c})$
    end for
    Sort(S_{parent}) // sort S_{parent} in ascending order of $S_{\text{parent}}[k].norm$
    $\mathbf{z}_{\text{best}} := S_{\text{parent}}[1].\mathbf{z}$
    $norm_{\text{best}} := S_{\text{parent}}[1].norm$
    $counter := 0$

Algorithm 2 Our algorithm (continued)

```
    while counter < M do
        for k := 1, ..., N_population do
            repeat
                a := RandInt(1, N_population)
                b := RandInt(1, N_population)
                S_offspring[k].z := Crossover(S_parent[a], S_parent[b])
                S_offspring[k].z := Mutation(S_offspring[k], p_mutation)
                S_offspring[k] := LocalSearch(S_offspring[k], μ, c)
                cnt_localsearch := cnt_localsearch + 1
            until Sum(S_offspring[k].z) ≠ 0 // avoid the zero vector
        end for
        for k := N_population + 1, ..., N_offspring do
            S_offspring[k] := S_parent[k − N_population] // preserve N_elite elites in the last generation of population
        end for
        Sort(S_offspring) // sort S_offspring in ascending order of S_offspring[k].norm
        if norm_sh < 1.05·GH(L(B)) then return Decode(z_sh, B) // return an optimal solution
        end if
        if norm_best > S_offspring[1].norm then
            z_best := S_offspring[1].z
            norm_best := S_offspring[1].norm
            counter := 0
        else
            counter := counter + 1
        end if
        S_parent[1] := S_offspring[1]
        kk := 2
        for k := 2, ..., N_population do
            if S_offspring[kk].norm ≠ S_parent[k − 1] then
                S_parent[k] := S_offspring[kk]
            else
                k := k − 1
            end if
            kk := kk + 1
            if kk = N_offspring then
                counter := M
                break
            end if
        end for
    end while
    if d > 0 then
        sum_i_in := sum_i_in + i_in
        m_in := sum_i_in/(cnt_restart + 1)
        if i_in < m_in then
            cnt_i_in_s := cnt_i_in_s + 1
        else
            cnt_i_in_l := cnt_i_in_l + 1
        end if
        if cnt_i_in_s ≠ 0 then
            r_in := cnt_i_in_l/cnt_i_in_s
        end if
        v_in := Decode(z_in, μ, B)
        G := [b_1, ..., b_{i_in−1}, v_in, b_{i_in+1}, ..., b_n]
        B := BKZ(G, β)
        B := S²LLL(B)
    end if
    gs_sum_prev := gs_sum
    cnt_restart := cnt_restart + 1
end while
```

Algorithm 3 Initialization

Input:

t: a parameter of NNR.

$N_{\text{population}}$: the parent population size.

1: **for** $k := 1, \ldots, N_{\text{population}}$ **do**
2: **for** $j := 0, \ldots, \mathbf{t}.size() - 2$ **do**
3: **for** $i := \mathbf{t}[j] + 1, \ldots, \mathbf{t}[j+1]$ **do**
4: **if** $i \leq j0$ **then**
5: $S_{\text{parent}}[k].\mathbf{z}[i] :=$ RandInt(1, j) // RandInt(1, j) returns a random integer uniformly distributed in the range [1, j], including both end points
6: **else**
7: $S_{\text{parent}}[k].\mathbf{z}[i] := 0$
8: **end if**
9: **end for**
10: **end for**
11: **end for**

Algorithm 4 InitializationOuter

Input:
M: the restart strategy parameter
$\mathbf{t}[1]$: the second element of a parameter of NNR
r_{th}: the threshold of r_{in}
$N_{\text{population}}$: the parent population size
N_{elite}: the elite population size
$j0$: the maximum index of the last nonzero element of NNR
variables defined in this step:
$cnt_restart$: a counter variable used to keep track of the number of times a restart run has been called
ev: the evolution velocity
$S_{\text{parent_}M}$: a structure array, each element of which consists of a binary string that is a candidate chromosome for an optimal value of M and the value of ev linked to the chromosome.
$S_{\text{offspring_}M}$: a two-dimensional int array, each element of which consists of a binary string (int_array) $\mathbf{z}$ that is an individual for the next generation.
/* Here, descriptions of definitions of the structure arrays for other parameters are omitted. */

1: $cnt_restart := 0$
2: // initialization of the chromosomes corresponding to M
3: $len_M := \text{BN}(M) + 3$
4: $N_{\text{population_}M} := 2len_M$
5: $N_{\text{elite_}M} := len_M$
6: $S_{\text{parent_}M} := \text{Malloc}(\text{parameter}, N_{\text{population_}M})$
7: $\text{Copy}(S_{\text{parent_}M}[1].\mathbf{z}, \text{BS}(M, len_M))$
8: **for** $k := 2, \ldots, N_{\text{population_}M}$ **do**
9: $\quad \text{Copy}(S_{\text{parent_}M}[k].\mathbf{z}, \text{BS}(\text{Floor}(M + \text{Rand}() \cdot 2len_M - len_M + 0.5), len_M))$
10: **end for**
11: **for** $k := 1, \ldots, N_{\text{population_}M}$ **do**
12: $\quad S_{\text{parent_}M}[k].\mathbf{z}[1] := 1$
13: **end for**
14: $S_{\text{offspring_}M} := \text{Malloc}(\text{int_array}, N_{\text{population_}M})$
15: **for** $k := 1, \ldots, N_{\text{population_}M}$ **do**
16: $\quad S_{\text{offspring_}M}[k].\mathbf{z} := \text{Malloc}(\text{int}, len_M)$
17: **end for**

Algorithm 4 InitializationOuter (continued)

/* Initializations of the chromosomes corresponding to other parameters are described, but the same description patterns (consequently, most of the descriptions) are omitted. Here, "$\cdots$" is the symbol for the same description patterns. */

// initialization of the chromosomes corresponding to $N_{\text{population}}$
$len_{N_{\text{population}}}$:= BN($N_{\text{population}}$) + 1
$\cdots$
for $k := 1, \ldots, N_{\text{population}_N_{\text{population}}}$ **do**
    $S_{\text{parent}_N_{\text{population}}}[k].\mathbf{z}[1] := 1$
end for
$\cdots$
// initialization of the chromosomes corresponding to N_{elite}
$len_{N_{\text{elite}}}$:= BN(N_{elite}) + 1
$\cdots$
for $k := 1, \ldots, N_{\text{population}_N_{\text{elite}}}$ **do**
    $S_{\text{parent}_N_{\text{elite}}}[k].\mathbf{z}[1] := 1$
end for
$\cdots$
// initialization of the chromosomes corresponding to $\mathbf{t}_1$
$len_{\mathbf{t}_1}$:= BN($\mathbf{t}_1$)
$\cdots$
// initialization of the chromosomes corresponding to p_{mutation}
bn := BN(n) − 1
$len_{p_{\text{mutation}}} := bn + 4$
$\cdots$
$S_{\text{parent}_p_{\text{mutation}}}$:= Malloc(int_array, $N_{\text{population}_p_{\text{mutation}}}$)
for $k := 1, \ldots, N_{\text{population}_p_{\text{mutation}}}$ **do**
    $S_{\text{parent}_p_{\text{mutation}}}[k].\mathbf{z}$:= Malloc(int, $len_{p_{\text{mutation}}}$)
end for
for $i := 1, \ldots, len_{p_{\text{mutation}}}$ **do**
    $S_{\text{parent}_p_{\text{mutation}}}[1].\mathbf{z}[i] := 0$
end for
$S_{\text{parent}_p_{\text{mutation}}}[1].\mathbf{z}[bn] := 1$
for $k := 2, \ldots, N_{\text{population}_p_{\text{mutation}}}$ **do**
    for $i := 1, \ldots, len_{p_{\text{mutation}}}$ **do**
        if $i < bn$ **then**
            $S_{\text{parent}_p_{\text{mutation}}}[k].\mathbf{z}[i] := S_{\text{parent}_p_{\text{mutation}}}[1].\mathbf{z}[i]$
        else if Rand() < 0.5 **then**
            $S_{\text{parent}_p_{\text{mutation}}}[k].\mathbf{z}[i] := S_{\text{parent}_p_{\text{mutation}}}[1].\mathbf{z}[i]$
        else
            $S_{\text{parent}_p_{\text{mutation}}}[k].\mathbf{z}[i] :=\! !S_{\text{parent}_p_{\text{mutation}}}[1].\mathbf{z}[i]$
        end if
    end for
end for
$\cdots$

Algorithm 4 InitializationOuter (continued)

// initialization of the chromosomes corresponding to r_{th}
$len_{r_{th}}$:= FBN(r_{th}) + 7
$\cdots$
$S_{\text{parent}_r_{th}}$:= Malloc(int_array, $N_{\text{population}_r_{th}}$)
for $k := 2, \ldots, N_{\text{population}_r_{th}}$ **do**
    $S_{\text{parent}_r_{th}}[k].\mathbf{z}$:= Malloc(int, $len_{r_{th}}$)
end for
Copy($S_{\text{parent}_r_{th}}[1].\mathbf{z}$, FBS($r_{th}, len_{r_{th}}$))
dp := BN(Floor(r_{th})) // the binary length of the integer part
for $k := 2, \ldots, N_{\text{population}_r_{th}}$ **do**
    for $i := 1, \ldots, len_{r_{th}}$ **do**
        if $i \leq dp$ **then**
            $S_{\text{parent}\,r_{th}}[k].\mathbf{z}[i] := S_{\text{parent}_r_{th}}[1].\mathbf{z}[i]$
        else if Rand() < 0.5 **then**
            $S_{\text{parent}_r_{th}}[k].\mathbf{z}[i] := S_{\text{parent}_r_{th}}[1].\mathbf{z}[i]$
        else
            $S_{\text{parent}_r_{th}}[k].\mathbf{z}[i] := !S_{\text{parent}_r_{th}}[1].\mathbf{z}[i]$
        end if
    end for
end for
$\cdots$
// initialization of the chromosomes corresponding to $j0$
len_{j0} := BN($j0$)
$\cdots$

Algorithm 5 FitnessEvaluation [13]

Input:
z: a chromosome.
$\boldsymbol{\mu}$: the coefficients of the Gram–Schmidt orthogonalization process.
c: the squared Euclidean norms of the Gram-Schmidt orthogonalized vectors.
Output:
norm: the squared Euclidean norm of the lattice vector that corresponds to **z**.

for $i := n, \ldots, 1$ **do**
    // the index i decreases from n to 1
    if $\mathbf{z}[i] \neq 0$ **then**
        $l := i$ // l is the index of the last nonzero entry of **z**
        break
    end if
end for
for $i := 1, \ldots, l$ **do**
    $\boldsymbol{\nu}[i] := \text{Ceil}(\mathbf{z}[l]/2) \cdot \boldsymbol{\mu}[l][i]$
end for
$norm := norm + \boldsymbol{\nu}[l] \cdot \boldsymbol{\nu}[l] \cdot \mathbf{c}[l]$
for $i := l, \ldots, 1$ **do**
    // the index i decreases from l to 1
    $\mathbf{t}[i] := \text{Ceil}(\boldsymbol{\nu}[i] - 0.5)$
    if $\mathbf{t}[i] < \boldsymbol{\nu}[i]$ **then**
        $\mathbf{t}[i] := \mathbf{t}[i] - (-1)^{\mathbf{z}[i]}\text{Ceil}(\mathbf{z}[i]/2)$
    elset$[i] := \mathbf{t}[i] + (-1)^{\mathbf{z}[i]}\text{Ceil}(\mathbf{z}[i]/2)$
    end if
    for $j := 1, \ldots, i$ **do**
        $\boldsymbol{\nu}[j] := \boldsymbol{\nu}[j] - \mathbf{t}[i] \cdot \boldsymbol{\mu}[i][j]$
    end for
    $norm := norm + \boldsymbol{\nu}[i] \cdot \boldsymbol{\nu}[i] \cdot \mathbf{c}[i]$
end for

Algorithm 6 FitnessEvaluationEX

Input:
z: a chromosome
$\boldsymbol{\mu}$: the coefficients of the Gram–Schmidt orthogonalization process
c: the squared Euclidean norms of the Gram-Schmidt orthogonalized vectors
Output:
norm: the squared Euclidean norm of the lattice vector which corresponds to **z**

for $i := n, \ldots, 1$ **do**
  // the index i decreases from n to 1
  if $\mathbf{z}[i] \neq 0$ **then**
    $l := i$ // l is the index of the last nonzero entry of **z**
    break
  end if
end for
for $i := 1, \ldots, l$ **do**
  $\boldsymbol{\nu}[i] := \mathrm{Ceil}(\mathbf{z}[l]/2) \cdot \boldsymbol{\mu}[l][i]$
end for
$norm := norm + \boldsymbol{\nu}[l] \cdot \boldsymbol{\nu}[l] \cdot \mathbf{c}[l]$
for $i := l, \ldots, 1$ **do**
  // the index i decreases from l to 1
  $\mathbf{t}[i] := \mathrm{Ceil}(\boldsymbol{\nu}[i] - 0.5)$
  if $\mathbf{t}[i] < \boldsymbol{\nu}[i]$ **then**
    $\mathbf{t}[i] := \mathbf{t}[i] - (-1)^{\mathbf{z}[i]}\mathrm{Ceil}(\mathbf{z}[i]/2)$
  elset$[i] := \mathbf{t}[i] + (-1)^{\mathbf{z}[i]}\mathrm{Ceil}(\mathbf{z}[i]/2)$
  end if
  for $j := 1, \ldots, i$ **do**
    $\boldsymbol{\nu}[j] := \boldsymbol{\nu}[j] - \mathbf{t}[i] \cdot \boldsymbol{\mu}[i][j]$
  end for
  $norm := norm + \boldsymbol{\nu}[i] \cdot \boldsymbol{\nu}[i] \cdot \mathbf{c}[i]$
  if $cnt_restart \leq 100$ **or** $r_{in} \leq r_{th}$ **or** $i \leq m_i$ **then**
    if $is_candidate \neq 0$ **and** $0 < \mathbf{c}[i] - norm$ **then**
      $d := d'$
      $i_{in} := i'_{in}$
      for $j := 1, \ldots, n$ **do**
        $\mathbf{z}_{in}[j] := \mathbf{z}'_{in}[j]$
      end for
      $is_candidate := 0$
    end if
    if $0 < norm$ **and** $d < \mathbf{c}[i] - norm$ **then**
      $d' := d$
      $i'_{in} := i_{in}$
      for $j := 1, \ldots, n$ **do**
        $\mathbf{z}'_{in}[j] := \mathbf{z}_{in}[j]$
      end for
      $d := \mathbf{c}[i] - norm$
      $i_{in} := i$
      for $j := 1, \ldots, n$ **do**
        $\mathbf{z}_{in}[j] := \mathbf{z}[j]$
      end for
      $is_candidate := 1$
    end if
    if $i = 1$ **and** $0 < norm < norm_sh$ **then**
      $norm_sh := norm$
      for $j := 1, \ldots, n$ **do**
        $\mathbf{z}_{sh}[j] := \mathbf{z}[j]$
      end for
    end if
  end if
  $is_candidate := 0$
end for

Algorithm 7 Crossover [13]

Input:

$S_{\text{parent}}[a], S_{\text{parent}}[b]$: randomly selected parents for mating.

Output:

$\mathbf{z}_{\text{offspring}}$: a new chromosome.

for $i := 1, \ldots,$ $\mathbf{t}[1]$ **do**
    $\mathbf{z}_{\text{offspring}}[i] := 0$ // set all initial entries in $\mathbf{z}_{\text{offspring}}$ to 0; by "initial entries", we refer to those $\mathbf{z}_{\text{offspring}}[i]$ whose indices are range between 1 and $\mathbf{t}[1]$.
end for
for $i := \mathbf{t}[1] + 1, \ldots, n$ **do**
    if Rand() < 0.5 **then**
        $\mathbf{z}_{\text{offspring}}[i] := S_{\text{parent}}[a].\mathbf{z}[i]$
    else
        $\mathbf{z}_{\text{offspring}}[i] := S_{\text{parent}}[b].\mathbf{z}[i]$
    end if
end for

Algorithm 8 Mutation [13]

Input:

$S_{\text{offspring}}[k]$: an offspring of a pair of parents.

p_{mutation}: a mutation rate.

Output:

$\mathbf{z}_{\text{offspring}}$: a mutated chromosome.

for $i := \mathbf{t}[1] + 1, \ldots, n$ **do**
    // mutate each of the entries in $S_{\text{offspring}}[k].\mathbf{z}$ except the initial ones
    $r :=$ Rand()
    if $r < p_{\text{mutation}}$ **then**
        $r_{\text{snd}} :=$ RandSND() // RandSND() returns a random number drawn from a standard normal distribution
        $\mathbf{z}_{\text{offspring}}[i] := \text{Max}(S_{\text{offspring}}[k].\mathbf{z}[i], S_{\text{offspring}}[k].\mathbf{z}[i] + \text{RI}(r_{\text{snd}}))$ // $\text{RI}(r_{\text{snd}})$ rounds r_{snd} toward infinity, i.e., returns $\text{Sign}(r_{\text{snd}}) \cdot \lceil r_{\text{snd}} \rceil$
    else
        $\mathbf{z}_{\text{offspring}}[i] := S_{\text{offspring}}[k].\mathbf{z}[i]$
    end if
end for

Algorithm 9 LocalSearch

Input:
$S_{\text{offspring}}[k]$: an individual.
$\boldsymbol{\mu}$: the coefficients of the Gram–Schmidt orthogonalization process.
$\mathbf{c}$: the squared Euclidean norms of the Gram-Schmidt orthogonalized vectors.
Output:
$S_{\text{offspring}}[k]$: a better individual.

for $i := 1, \ldots, n$ **do**
    $\mathbf{z}''[i] := S_{\text{offspring}}[k].\mathbf{z}[i]$
end for
$norm'' :=$ FitnessEvaluation($\mathbf{z}'', \boldsymbol{\mu}, \mathbf{c}$)
$is_found := 1$
while $is_found \neq 0$ **do**
    $is_found := 0$
    for $i := \mathbf{t}[1] + 1, \ldots, n$ **do**
        // narrow the search space by fixing the initial entries in $S_{\text{offspring}}[k].\mathbf{z}$ to 0
        for $j := 1, \ldots, n$ **do**
            $\mathbf{z}'[j] := S_{\text{offspring}}[k].\mathbf{z}[j]$
        end for
        $\mathbf{z}'[i] :=!\mathbf{z}'[i]$ // switch 0 and a nonzero gene to 1 and 0, respectively
        $norm' :=$ FitnessEvaluationEX($\mathbf{z}', \boldsymbol{\mu}, \mathbf{c}$)
        if $0 < norm' < norm''$ **then**
            for $j := 1, \ldots, n$ **do**
                $\mathbf{z}''[j] := \mathbf{z}'[j]$
            end for
            $norm'' := norm'$
            $is_found := 1$
        end if
    end for
    for $i := 1, \ldots, n$ **do**
        $S_{\text{offspring}}[k].\mathbf{z}[i] := \mathbf{z}''[i]$
    end for
    $S_{\text{offspring}}[k].norm := norm''$
end while

Algorithm 10 Decode [13]

Input:
$\mathbf{z}$: a chromosome.
$\boldsymbol{\mu}$: the coefficients of the Gram–Schmidt orthogonalization process.
B: a lattice basis $[\mathbf{b}_1, \ldots, \mathbf{b}_n]$.
Output:
$\mathbf{v}$: a lattice vector.

for $i := n, \ldots, 1$ **do**
    // the index i decreases from n to 1
    $\mathbf{y}[i] := 0$
    for $j := i + 1, \ldots, n$ **do**
        $\mathbf{y}[i] := \mathbf{y}[i] - \mathbf{u}[j] \cdot \boldsymbol{\mu}[j][i]$
    end for
    $\mathbf{u}[i] := \text{Floor}(\mathbf{y}[i] + 0.5)$
    if $\mathbf{u}[i] < \mathbf{y}[i]$ **then**
        $\mathbf{u}[i] := \mathbf{u}[i] - (-1)^{\mathbf{z}[i]}\text{Ceil}(\mathbf{z}[i]/2)$
    else $\mathbf{u}[i] := \mathbf{u}[i] + (-1)^{\mathbf{z}[i]}\text{Ceil}(\mathbf{z}[i]/2)$
    end if
end for
$\mathbf{v} := \text{Mul}(B, \mathbf{u})$ // $\text{Mul}(B, \mathbf{u})$ returns the product of B and $\mathbf{u}$

Algorithm 11 OuterGA

Input:
$\mathbf{t}[1]$: the second element of a parameter of NNR
$j0$: the maximum index of the last nonzero element of NNR

$ev := \text{Abs}(gs_sum - gs_sum_{prev})/(cnt_localsearch \cdot (j0 - \mathbf{t}[1] + 1))$
// a series of genetic operations for the chromosomes corresponding to M
$k_m := (cnt_restart - 1)\%N_{\text{population_}M} + 1$
$S_{\text{parent_}M}[k_m].ev := ev$
if $k_m = N_{\text{population_}M}$ **then**
    $\text{Sort}(S_{\text{parent_}M})$ // sort $S_{\text{parent_}M}$ in descending order of $S_{\text{parent_}M}[k].ev$
    for $k := 1, \ldots, N_{\text{elite_}M}$ **do**
        for $i := 1, \ldots, len_M$ **do**
            $S_{\text{offspring_}M}[k].\mathbf{z}[i] := S_{\text{parent_}M}[k].\mathbf{z}[i]$ // preserve $N_{\text{elite_}M}$ elites in the last generation of population
        end for
    end for
    for $k := N_{\text{elite_}M} + 1, \ldots, N_{\text{population_}M}$ **do**
        // selection
        $a := \text{RandInt}(1, N_{\text{population_}M})$
        $b := \text{RandInt}(1, N_{\text{population_}M})$
        // crossover
        for $i := 1, \ldots, len_M$ **do**
            $r := \text{Rand}()$
            if $r < 0.5$ **then**
                $S_{\text{offspring_}M}[k].\mathbf{z}[i] := S_{\text{parent_}M}[a].\mathbf{z}[i]$
            else
                $S_{\text{offspring_}M}[k].\mathbf{z}[i] := S_{\text{parent_}M}[b].\mathbf{z}[i]$
            end if
        end for
        // mutation
        for $i := 2, \ldots, len_M$ **do**
            $r := \text{Rand}()$
            if $r < 1/N_{\text{population_}M}$ **then**
                $S_{\text{offspring_}M}[k].\mathbf{z}[i] :=!S_{\text{offspring_}M}[k].\mathbf{z}[i]$ // switch 0 to 1, and vice versa
            end if
        end for
    end for
    for $k := 1, \ldots, N_{\text{population_}M}$ **do**
        $S_{\text{parent_}M}[k] := S_{\text{offspring_}M}[k]$ // form the new generation
    end for
end if
$M := \text{Dec}(S_{\text{parent_}M}[(k_m + 1)\%N_{\text{population_}M}])$
// $\text{Dec}(S_{\text{parent_}M}[(k_m + 1)\%N_{\text{population_}M}])$ returns the decimal number corresponding to the binary string $S_{\text{parent_}M}[(k_m + 1)\%N_{\text{population_}M}]$

Algorithm 11 Outer GA (continued)

/* The parameter selections based on the outer GA for other parameters are described, but the same description patterns (consequently, most of the descriptions) are omitted. Here, "$\cdots$" is the symbol for the same description patterns. */

// a series of genetic operations for the chromosomes corresponding to $N_{\text{population}}$
$k_{N_{\text{population}}} := (cnt_restart - 1)\%N_{\text{population_}N_{\text{population}}} + 1$
$\cdots$
    // mutation
    for $i := 2, \ldots, len_{N_{\text{population}}}$ **do**
$\cdots$
// a series of genetic operations for the chromosomes corresponding to N_{elite}
$k_{N_{\text{elite}}} := (cnt_restart - 1)\%N_{\text{population_}N_{\text{elite}}} + 1$
$\cdots$
    // mutation
    for $i := 2, \ldots, len_{N_{\text{elite}}}$ **do**
$\cdots$
$N_{\text{elite}} := \text{Dec}(S_{\text{parent_}N_{\text{elite}}}[(k_{N_{\text{elite}}} + 1)\%N_{\text{population_}N_{\text{elite}}}])$
if $N_{\text{elite}} > N_{\text{population}}$ **then**
    $N_{\text{elite}} := N_{\text{population}}$
    $\text{Copy}(S_{\text{parent_}N_{\text{elite}}}[(k_{N_{\text{elite}}} + 1)\%N_{\text{population_}N_{\text{elite}}}].\mathbf{z}, \text{BS}(N_{\text{elite}}, len_{N_{\text{elite}}}))$
end if
$N_{\text{offspring}} := N_{\text{population}} + N_{\text{elite}}$
// a series of genetic operations for the chromosomes corresponding to $\mathbf{t}_1$
$k_{\mathbf{t}_1} := (cnt_restart - 1)\%N_{\text{population_}\mathbf{t}_1} + 1$
$\cdots$
    // mutation
    for $i := 1, \ldots, len_{\mathbf{t}_1} - 2$ **do**
$\cdots$
// a series of genetic operations for the chromosomes corresponding to p_{mutation}
$k_{p_{\text{mutation}}} := (cnt_restart - 1)\%N_{\text{population_}p_{\text{mutation}}} + 1$
$\cdots$
    // mutation
    for $i := 6, \ldots, len_{p_{\text{mutation}}}$ **do**
$\cdots$
$p_{\text{mutation}} := \text{Dec}(S_{\text{parent_}p_{\text{mutation}}}[(k_{p_{\text{mutation}}} + 1)\%N_{\text{population_}p_{\text{mutation}}}])$
if $!p_{\text{mutation}}$ **then**
    $S_{\text{parent_}p_{\text{mutation}}}[(k_{p_{\text{mutation}}} + 1)\%N_{\text{population_}p_{\text{mutation}}}].\mathbf{z}[len_{p_{\text{mutation}}}] := 1$ // the least significant bit of p_{mutation} is switched to 1
    $p_{\text{mutation}} := \text{Dec}(S_{\text{parent_}p_{\text{mutation}}}[(k_{p_{\text{mutation}}} + 1)\%N_{\text{population_}p_{\text{mutation}}}])$
end if
// a series of genetic operations for the chromosomes corresponding to r_{th}
$k_{r_{th}} := (cnt_restart - 1)\%N_{\text{population_}r_{th}} + 1$
$\cdots$
    // mutation
    for $i := 3, \ldots, len_{r_{th}}$ **do**
$\cdots$
// a series of genetic operations for the chromosomes corresponding to $j0$
$k_{j0} := (cnt_restart - 1)\%N_{\text{population_}j0} + 1$
$\cdots$
    // mutation
    for $i := 1, \ldots, len_{j0} - 2$ **do**
$\cdots$
$j0 := \text{Dec}(S_{\text{parent_}j0}[(k_{j0} + 1)\%N_{\text{population_}j0}])$
if $j0 < n - 7$ **then**
    $j0 := n - 7$
    $\text{Copy}(S_{\text{parent}\,j0}[(k_{j0} + 1)\%N_{\text{population_}j0}].\mathbf{z}, \text{BS}(j0, len_{j0}))$
else if $j0 > n$ **then**
    $j0 := n$
    $\text{Copy}(S_{\text{parent}\,j0}[(k_{j0} + 1)\%N_{\text{population_}j0}].\mathbf{z}, \text{BS}(j0, len_{j0}))$
end if