A Genetic Algorithm for Searching Shortest Lattice Vector of SVP Challenge[⋆] [⋆⋆]

Dan Ding[1], Guizhen Zhu[2], Xiaoyun Wang[3,4]

[1] Department of Computer Science and Technology, Tsinghua University,
Beijing 100084, China P. R.
[2] Data Communication Science and Technology Research Institute,
Beijing 100191, China P. R.
[3] Institute for Advanced Study, Tsinghua University,
Beijing 100084, China P. R.
[4] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education,
Shandong University, Jinan 250100, China P. R.

Abstract. In this paper, we propose a genetic algorithm for solving the shortest vector problem (SVP) based on sparse integer representations of short vectors in lattices as chromesomes, which, we prove, can guarantee finding the shortest lattice vector under a Markov chain analysis. Moreover, we also suggest some improvements by introducing heuristic techniques: local search and heuristic pruning. The experimental results show that the genetic algorithm outperforms most enumeration algorithms in running time, and achieves the shortest vectors in larger dimensions under SVP challenge benchmarks[31].

Keywords: Shortest Vector Problem (SVP), Lattice-Based Cryptography, Genetic Algorithm, Chromesome, and Local Search.

1 Introduction

A lattice $\mathcal{L}$ is a discrete additive subgroup of $\mathbb{R}^m$. It can be generated by integral combinations of , say n, linear independent vectors $\mathbf{b}_1, \ldots, \mathbf{b}_n$ in $\mathbb{R}^m$, and the n vectors constitute the basis of the generated lattice. The discreteness of lattices implies that there exists a nonzero vector with the smallest non-zero Euclidean norm in each lattice, denoted as λ_1, and a lattice vector closest to a given target vector $\mathbf{t}$ in $\mathbb{R}^m$. Finding the two special lattice vectors leads to the two famous computational problems regarding lattices:

- Shortest Vector Problem (SVP): Given a lattice basis, find the shortest nonzero vector in the lattice;
- Closest Vector Problem (CVP): Given a lattice basis and a target vector, find the lattice vector closest to the target.

The CVP has been proved to be NP-complete by van Emde Boas [40] under Karp/Cook reduction in 1981 and refined by Micciancio and Goldwasser [23] in 2002. However, it is not known whether the SVP is NP-hard until 1998 when Ajtai [1] proved the NP-hardness of SVP through randomized reduction.

The two main problems SVP and CVP are of prime importance to the cryptography in recent years. A variety of public-key cryptosystems are proposed in recent years, and their security is as hard as solving the worst-case hardness of the variants of SVP and CVP, which paves a way for the proposal of many lattice-based cryptosystems[24]. These newly-proposed schemes are said to be promising candidates for the post-quantum cryptography, which are believed to be secure against the attack of the quantum computers, even though algorithms on quantum computers have made widely-used cryptosystems nowadays based on factorization and discrete logarithm insecure again as forwarded by Shor [34] in 1994. Therefore, the algorithms solving the two main problems are quite important for the research community.

[⋆] Supported by the National Natural Science Foundation of China (Grant No. 61133013)

[⋆⋆] Supported by 973 Program (Grant No. 2013CB834205)

During the last three decades, a number of algorithms aiming at solving the shortest (short) lattice vectors problems (SVP or SVP-variants) have been proposed, all of which are of the following two sorts in terms of the space complexity: the algorithms with polynomial space complexity and those with exponential space complexity. The formers include basis reduction algorithms including the celebrated **LLL** algorithm [20] which can find a vector as short as $2^{O(n)}\lambda_1$ within time polynomial in the rank n, and the Kannan-Helfrich's enumeration algorithm [16,14,17] which can find exactly the shortest vector, or the Hermite Korkin-Zolotarev basis, using a $n^{O(n)}$ time, and the blockwise Korkin-Zolotarev basis reduction[33,6] which can find a short vector subexponential to the shortest vector, maybe within subexponential time [13]. Furthermore, an enumeration algorithm with extreme pruning for SVP was presented by Gamma, Nguyen and Regev[11]. The second type of the algorithms include various sieve algorithms [2,28,25,41] which can find the shortest vector at a lattice in a high probability in $2^{O(n)}$ time and $2^{O(n)}$ space with the different constants in the index $O(n)$. And another deterministic algorithm based on Voronoi cell computation [25] which is able to find the shortest vector deterministically with exponential time $2^{2n+o(n)}$ and the space complexity $2^{n+o(n)}$.

In this paper, we devote to developing a new type of search algorithm based on the genetic algorithm, which also works on the polynomial space complexity. The genetic algorithm (GA) is originated by Holland [15] in 1975, which is, sometimes, called the Simple Generic Algorithm (SGA). The fundamental idea of the genetic algorithms is to mimic the evolutionary process in nature to search for the optimum solutions, and the main procedure of the simple genetic algorithm can be described as follows:

Table 1. The Simple Genetic Algorithm

```
Input: An initial population
Output: An optimum population including the optimum individual
1. Initialize population
2. While Optimum population not found do
   (a) Evaluate population by the fitness function
   (b) Selection: choose randomly individuals as mating parents
   (c) Perform Crossover and Mutation to generate new population
   (d) Evaluate new population
```

Ever since then, a variety of GAs have been developed and explored to address a hybrid of search or optimization problems such as learning or classifier system problems [12,5], combinatorial optimization problems [29,27,9], and knapsack problems [18,7], some of which are classical NP-hard problems.

As mentioned in the Table 1, the genetic algorithms include three basic components of operations: `crossover`, `mutation`, and `selection`. Moreover, various elitist strategies are also used, by GAs, to ensure them to converge to the optimal solutions. Actually, earlier elitist strategies[39][10] are adopted to meet conditions, under which GAs are guaranteed to converge. Among the analysis schemes, the Markov chain analysis, which models the process of genetic algorithms as a finite Markov chain, is an effective tool to prove rigorously the convergence of GAs as in [9,4,38,30]. For a survey of genetic algorithms, you can refer to [36,8].

Note that `crossover` and `mutation` are both bit operations, and it is necessary to encode each individual in a population into a chromesome (or, a bit string) for the genetic algorithms to operate on. Plenty of techniques have been adopted for encoding bit strings in genetic algorithms to solve different problems, some of which are quite delicate and natural. However, as far as Shortest Vector Problem (SVP) is concerned, it is a difficult task to encode a lattice vector into a chromesome for genetic algorithms to handle smoothly, which has not yet been researched into deeply.

A work[22] published in 2013 adopts the strategies of genetic algorithm to improve **LLL** reduction algorithm. However, even though their experimental results are reasonable, they didn't specify how they encoded lattice vectors (a **LLL**-reduced basis) into chromesomes, nor the details of their `crossover` or `mutation`. So we fail to obtain a full knowledge of their algorithm.

Our contributions in this paper are twofold. First, we discover sparse integer representations of the short vectors in lattices, which might be of independent interest to the existing SVP algorithms, and an appropriate way to encode lattice vectors into bit strings, as mentioned above. Second, we propose a genetic algorithm for the shortest lattice vector problem based on the sparse representations of short vectors. A Markov chain analysis of the genetic algorithm assures that it converges to, precisely, the shortest vector with a negligible probability of failure. Furthermore, we also present some improvements of our algorithm by adopting some heuristic techniques: local search and heuristic pruning. Experimental results show that the improved algorithm behaves fairly well on the benchmarks of the SVP challenge[31], generating short vectors, most of which are better than the previous challenge results, of lattices of dimensions up to 118, and a comparison of running times implies that the genetic algorithm outperforms present-day enumeration algorithms, like the Kannan-Helfrich enumeration and the enumeration with conservative pruning. We have posted 25 results on the SVP challenge since Feb. 2013. In summary, for the first time, we propose a genetic algorithm for the SVP, and its heuristically-revised version gains advantages over previous challenges in quality of generated vectors, and over other practical algorithms in their running times.

The rest of the paper is organized as follows: In Section 2, we provide some necessary backgrounds on lattices. In Section 3, we describe the sparse integer representations of the short vectors in lattices; Section 4 and 5 introduce the details of our genetic algorithm for SVP and its improvements. Finally, experimental results are given and compared in Section 6 and a conclusion in Section 7 follows.

2 Preliminaries

Let n be an integer, and let $\mathbb{R}^n$ be the n-dimensional Euclidean space with the inner product, denoted as $\langle \cdot, \cdot \rangle$, and the Euclidean norm of $\mathbf{v}$ is defined as $\|\mathbf{v}\| = \sqrt{\sum_{i=1}^{n} {v_i}^2}$, in which $\mathbf{v} = (v_1, \ldots, v_n) \in \mathbb{R}^n$. The closed sphere in $\mathbb{R}^n$ is denoted as $\mathcal{B}_n(\mathbf{O}, r)$ with $\mathbf{O}$ as its origin and r its radius. The linear space spanned by a set of vectors is denoted by $\text{span}(\cdot)$ and its orthogonal complement $\text{span}(\cdot)^{\perp}$, and $\mathbf{B}^T$ is the transpose of a matrix $\mathbf{B}$. We denote $\lfloor \cdot \rceil$ as the closest integer to a real number, and $\lfloor \cdot \rfloor$ as the closest integer less than or equal to a real number, while $\lceil \cdot \rceil$ the upper closest number.

2.1 Lattices

A *lattice* $\mathcal{L}$ is defined as the set of all integral combinations of n linear independent vectors $\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_n \in \mathbb{R}^m (m \geq n)$, where the vectors are referred to as the *basis* of the lattice, and n as its *rank*. If $m = n$, the lattice is called *full-rank*. All the lattices we discuss throughout this paper are full-rank lattices unless specified otherwise.

Conveniently, if given a matrix $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n] \in \mathbb{R}^{n \times n}$ with the n linear independent vectors as its columns, the lattice $\mathcal{L}$ generated by the basis $\mathbf{B}$ is defined as

$$\mathcal{L}(\mathbf{B}) = \{\mathbf{B}\mathbf{x} | \mathbf{x} \in \mathbb{Z}^n\} = \{\mathbf{v} \in \mathbb{R}^n \ | \mathbf{v} = \sum_{i=1}^{n} \mathbf{b}_i x_i, x_i \in \mathbb{Z}\}.$$

A single lattice can be generated by a series of the basis, or, in other words, the basis of one lattice is not unique. For a specific basis $\mathbf{B}$, the fundamental parallelepiped $\mathcal{P}(\mathbf{B})$ of the lattice is defined as:

$$\mathcal{P}(\mathbf{B}) = \{\mathbf{B}\mathbf{x} | \mathbf{x} = (x_1, \ldots, x_n) \in \mathbb{R}^n, \text{ for all } 0 \leq x_i < 1, i = 1, 2, \ldots, n\}.$$

The *determinant* $\det(\mathcal{L})$ of a lattice $\mathcal{L}$ is defined as the volume of the fundamental parallelepiped $\mathcal{P}(\mathbf{B})$ by selecting any basis $\mathbf{B}$. More precisely, for any basis $\mathbf{B}$ of a lattice $\mathcal{L}$, the determinant of $\mathcal{L}$ is computed as:

$$\det(\mathcal{L}) = \sqrt{\det(\mathbf{B}^T\mathbf{B})} = \sqrt{\det(\langle \mathbf{b}_i, \mathbf{b}_j \rangle)_{0 \leq i,j \leq n}}.$$

The determinant of a lattice is well-defined in the sense that the determinant does not depend on the choice of the basis. The i^{th} *successive minimum* $\lambda_i(\mathcal{L})$ of a lattice $\mathcal{L}$ implies the smallest radium of a sphere within which there are i linearly independent lattice points, i.e.,

$$\lambda_i(\mathcal{L}) = \inf\{r \in \mathbb{R}^n | \dim\{\text{span}(\mathcal{L} \cap \mathcal{B}_n(\mathbf{O}, r))\} = i\}.$$

For a basis $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ of a lattice $\mathcal{L}(\mathbf{B}) \in \mathbb{R}^{n\times n}$, its *Gram-Schmidt Orthogonalization* $\mathbf{B}^* = [\mathbf{b}_1^*, \dots, \mathbf{b}_n^*]$ is defined as,

$$\mathbf{b}_i^* = \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{ij}\mathbf{b}_j^*,$$

where

$$\mu_{ij} = \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}, \text{ for } 1 \leq j < i \leq n.$$

In other words, the Gram-Schmidt procedure projects $\mathbf{b}_i$ to the space orthogonal to the space spanned by $\mathbf{b}_1, \dots, \mathbf{b}_{i-1}$, and keeps the determinant unchanged, i.e., $\det(\mathbf{B}) = \prod_{i=1}^n \|\mathbf{b}_i^*\|$. For more details about lattices, refer to [23].

A basis $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ of a lattice $\mathcal{L}$ is ***LLL**-reduced* if it satisfies the following two conditions:

1. Its $|\mu_{ij}| \leq 1/2$, for $1 \leq j < i \leq n$;
2. and $\|\mathbf{b}_i + \mu_{i,i-1}\|^2 \geq \frac{3}{4}\|\mathbf{b}_{i-1}\|^2$, for $2 \leq i \leq n$.

In 1982, A. K. Lenstra and H. W. Lenstra and L. Lavász proposed the seminal **LLL** algorithm [20] that can reduce an arbitrary lattice basis into a **LLL**-reduced basis within a polynomial time complexity.

2.2 Korkin-Zolotarev Basis and Blockwise Korkin-Zolotarev Basis

Let $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ be a basis of a lattice $\mathcal{L} \subset \mathbb{R}^n$. Define $\pi_i : \mathbb{R}^n \mapsto \text{span}(\mathbf{b}_1, \dots, \mathbf{b}_{i-1})^\perp$ as the projection on the orthogonal complement of the span of the first $i-1$ bases of $\mathbf{B}$, for all $i \in \{1, 2, \dots, n\}$. $\pi_i(\mathbf{b}_j)$ is expressed as,

$$\pi_i(\mathbf{b}_j) = \mathbf{b}_j^* + \sum_{k=i}^{j-1} \mu_{jk}\mathbf{b}_k^*, \quad \text{if} \quad i < j.$$

In particular, $\pi_i(\mathbf{b}_i) = \mathbf{b}_i^*$ and $\pi_i(\mathbf{b}_j) = 0$ for $j < i$.

$\mathcal{L}_i^{(k)}$ is the lattice of rank k generated by the basis $[\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_{i+k-1})]$ in which $i + k \leq n + 1$. Clearly, it is true that $\mathcal{L}_i^{(n-i+1)} = \pi_i(\mathcal{L})$, which implies the lattice of rank $n - i + 1$ generated by basis $[\pi_i(\mathbf{b}_i), \dots, \pi_i(\mathbf{b}_n)]$.

In terms of the denotations aforementioned, a basis $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ is defined as *reduced in the sense of Korkin and Zolatarev* or *Korkin-Zolatarev basis*, or ***HKZ**-reduced basis*, if it satisfies that:

1. Its $|\mu_{ij}| \leq 1/2$, for $1 \leq j < i \leq n$;
2. and $\pi_i(\mathbf{b}_i)$ is the shortest vector of the lattice $\mathcal{L}_i^{(n-i+1)}$ under the Euclidean norm, for $1 \leq i \leq n$.

Similarly, a basis $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ is defined as a *β-blockwise Korkin-Zolotarev basis*, or ***BKZ**-reduced basis*, if the following conditions hold:

1. Its $|\mu_{ij}| \leq 1/2$, for $1 \leq j < i \leq n$;
2. and $\pi_i(\mathbf{b}_i)$ is the shortest vector of the lattice $\mathcal{L}_i^{(\min(\beta, n-i+1))}$ under the Euclidean norm, for $1 \leq i \leq n$.

2.3 The Blockwise Korkin-Zolotarev (BKZ) Algorithm

The Blockwise-Korkin-Zolotarev (**BKZ**) algorithm[32,33], on an input lattice basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ of rank n, outputs its **BKZ**-reduced basis with a blocksize β and a reduction factor ϵ, in which $2 \leq \beta \leq n$ and $0 < \epsilon \leq 1$. ϵ is always fixed for **LLL** reduction, and so we call the algorithm as β-**BKZ** algorithm.

As in Table 2, the algorithm starts with a **LLL**-reduced basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, and recursively reduces each projected sublattices $\mathcal{L}_i^{(\min(\beta,n-i+1))}$ for i to run through $[1, \ldots, n]$, such that the the first vectors of each such sublattices is its shortest vectors, i. e. $\|\pi_i(\mathbf{b}_i)\| = \lambda_1(\mathcal{L}_i^{(\min(\beta,n-i+1))})$ for all $1 \leq i \leq n$. The algorithm invokes an enumeration procedure **ENU** to find such shortest vector in each sublaticces of rank β, which is most time-consuming part and, so far, several improved methods[6,16,14,11] has been proposed to accelerate the **BKZ** algorithm.

Table 2. β-**BKZ Algorithm**

```
Input: A lattice basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$, its Gram-Schmidt Orthogonalization $\mathbf{B}^* = [\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$
as well as its $\mu = \{\mu_{i,j}\}$, and blocksize $\beta$.
Output: The $\beta$-BKZ reduced basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$
1. $z \leftarrow 0$; $i \leftarrow 1$; LLL$(\mathbf{b}_1, \ldots, \mathbf{b}_n, \mu)$;// LLL-reduce the basis, and update $\mu$
2. While $z < n - 1$ do
  (a) $i \leftarrow (i \mod (n-1)) + 1$; $k \leftarrow \min(i + \beta - 1, n)$; $h \leftarrow \min(k + 1, n)$;// define the current
projected sublattices $\mathcal{L}_i^{(\min(\beta,n-i+1))}$
  (b) $\mathbf{x} \leftarrow$ ENU$(\|\mathbf{b}_i^*\|^2, \ldots, \|\mathbf{b}_k^*\|^2, \mu)$;// find $\mathbf{x} = [x_i, \ldots, x_k]$ such that $\left\|\sum_{j=i}^{k} \mathbf{b}_j x_i\right\| =$
$\lambda_1(\mathcal{L}_i^{(\min(\beta,n-i+1))}$
  (c) If $\mathbf{x} \neq (1, 0, \ldots, 0)$ Then
        $z \leftarrow 0$; LLL$(\mathbf{b}_1, \ldots, \mathbf{b}_{i-1}, \sum_{j=i}^{k} \mathbf{b}_j x_j, \ldots, \mathbf{b}_h, \mu)$ at stage i;//insert the new vector at
the start of current sublattice, LLL-reduce it ro remove its linear dependency, and update $\mu$
  (d) Else
        $z \leftarrow z + 1$; LLL$(\mathbf{b}_1, \ldots, \mathbf{b}_h, \mu)$ at stage $h - 1$;// LLL-reduce the next sublattice
before enumeration
3. Return B which is $\beta$-BKZ reduced.
```

3 *y*-Sparse Representations of Lattice Vectors: Lattice Vectors from Another Point of View

Before devoting to the genetic algorithm, we should encode a lattice vector into a bit sequence, or, namely, a *chromesome* in genetic algoithms, since genetic algorithms can only operate on bit sequences. However, the elements of lattice vectors are real numbers, which defies encoding into bit strings. In order to address this problem, we need to find a method to transform the lattice vectors into bit strings. Given a certain basis $\mathbf{B}$ of a lattice, a lattice vector $\mathbf{v}$ can be represented as the integral combination of the basis vectors $\mathbf{v} = \mathbf{B}\mathbf{x}$, in which elements of the coordinate $\mathbf{x}$ are all integers. Therefore, we can use the coordinate $\mathbf{x}$ to generate the *chromesome*: transform every the elements of the vector into signed binary integers (with one bit as the sign) and concatenate them in order together. But there is one drawback in this method: it is difficult to bound the bit length of each element of $\mathbf{x}$ for the short vectors, and, then, the length of the bit string might be quite long. So, in this section, we change $\mathbf{x}$ into another form of integer vector $\mathbf{y}$ which can avoid the drawback of $\mathbf{x}$ above.

3.1 An Equivalent Integer Representations of Lattice Vectors Relating to Gram-Schmidt Orthorgonalization

In this subsection, we lay no assumptions on the lattice basis $[\mathbf{b}_1, \dots, \mathbf{b}_n]$, or, in other words, the basis is arbitrary. Given a lattice $\mathcal{L}(\mathbf{B})$, any vector $\mathbf{v}$ in this lattice can be represented as:

$$\mathbf{v} = \mathbf{B}\mathbf{x} = \sum_{i=1}^{n} x_i \mathbf{b}_i,$$

in which $\mathbf{x} = (x_1, \dots, x_n) \in \mathbb{Z}^n$. Putting it in another way, we can represent the vector $\mathbf{v}$ in terms of the Gram-Schmidt orthorgonalization $\mathbf{B}^*$ in place of $\mathbf{B}$ as described in the following.

Definition 1. *Given a lattice basis* $\mathbf{B} = [\mathbf{b}_1, \dots, \mathbf{b}_n]$ *and its Gram-Schimdt orthogonalization* $\mathbf{B}^* = [\mathbf{b}_1^*, \dots, \mathbf{b}_n^*]$ *with the matrix* $\mathbf{R} = [\mu_{ij}]_{1 \le i,j \le n} \in \mathbb{R}^{n \times n}$ *such that* $\mathbf{B} = \mathbf{B}^*\mathbf{R}$*, any vector* $\mathbf{v} \in \mathcal{L}(\mathbf{B})$*, and* $\mathbf{v} = \mathbf{B}\mathbf{x}$*, in which* $\mathbf{x} = (x_1, x_2, \dots, x_n)$*, we define a vector* $\mathbf{t} = (t_1, \dots, t_n) \in \mathbb{R}^n$ *as*

$$t_i = \begin{cases} 0 & \textit{for } i = n, \\ \sum_{j=i+1}^{n} \mu_{ji} x_j & \textit{for } i < n. \end{cases}$$

and another vector $\mathbf{y} = (y_1, y_2, \dots, y_n) \in \mathbb{Z}^n$ *such that,*

$$y_i = \lfloor x_i + t_i \rceil$$

Remark 1. Since $x_i \in \mathbb{Z}$, we have

$$y_i = x_i + \lfloor t_i \rceil (1 \le i \le n),$$

and thus,

$$x_i = y_i - \lfloor t_i \rceil (1 \le i \le n).$$

Thereby establishing a one-to-one correspondence between $\mathbf{x}$ and $\mathbf{y}$, and, also, a one-to-one correspondence between $\mathbf{v}$ and $\mathbf{y}$ as described below:

$$\mathbf{y} \xleftrightarrow{\mathbf{y}=\mathbf{x}+\lfloor \mathbf{t} \rceil} \mathbf{x} \xleftrightarrow{\mathbf{v}=\mathbf{B}\mathbf{x}} \mathbf{v},$$

where $\lfloor \mathbf{t} \rceil = (\lfloor t_1 \rceil, \dots, \lfloor t_n \rceil) \in \mathbb{Z}^n$. Therefore, we call the $\mathbf{y}$ corresponding to a lattice vector $\mathbf{v}$ as $\mathbf{y} \sim \mathbf{v}$.

This is quite important to represent a lattice vector as a chromesome in the genetic algorithm to be described in the next section.

For any vector $\mathbf{v}$ in the lattice $\mathcal{L}(\mathbf{B})$, the new integer vector $\mathbf{y} = (y_1, \dots, y_n) \in \mathbb{Z}^n$ described above satisfies that:

$$\mathbf{v} = \sum_{i=1}^{n} (y_i + \epsilon_i) \mathbf{b}_i^*,$$

in which $0 \le |\epsilon_i| \le 1/2$.

Then,

$$\sum_{y_i \neq 0} \left(|y_i| - \frac{1}{2} \right)^2 \|\mathbf{b}_i^*\|^2 \le \|\mathbf{v}\|^2 \le \sum_{i=1}^{n} \left(|y_i| + \frac{1}{2} \right)^2 \|\mathbf{b}_i^*\|^2. \tag{1}$$

Provided that $\mathbf{y}$ is the integer vector relating to the shortest vector $\mathbf{v}(\|\mathbf{v}\| = \lambda_1(\mathcal{L}))$, it satisfies:

$$\sum_{y_i \neq 0} \left(|y_i| - \frac{1}{2} \right)^2 \|\mathbf{b}_i^*\|^2 \le \|\mathbf{v}\|^2.$$

Since $\|\mathbf{v}\| = \lambda_1(\mathcal{L})$ and $y_i \in \mathbb{Z}$ for $1 \le i \le n$, we have

$$\sum_{i=1}^{n} \Big(\frac{1}{2}|y_i|\Big)^2 \|b_i^*\|^2 \le \sum_{y_i \ne 0} \Big(|y_i| - \frac{1}{2}\Big)^2 \|\mathbf{b}_i^*\|^2 \le \|\mathbf{v}\|^2 = \lambda_1^2(\mathcal{L}).$$

That is to say,

$$\sum_{i=1}^{n} |y_i|^2 \, \|b_i^*\|^2 \le 4\lambda_1^2(\mathcal{L}). \tag{2}$$

This means that, we can bound the the bit-length of $\mathbf{y}$ by guessing the upper bound of λ_1. The left is to find an appropriate Gram-Schimdt orthogonalization, and decrease the length of y as small as possible. One choice is to find a good Gram-Schimdt orthogonalization by invoking the **BKZ**-reduced basis.

3.2 The Estimate of y under β-Blockwise Korkin-Zolotarev Basis

Rewriting the the Proposition 4.2 in [19] gives the following lemma.

Lemma 1. *Let $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ be a Korkin-Zolotarev basis of a lattice $\mathcal{L}(\mathbf{B})$, with Gram-Schmidt orthogonalization $[\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$. Then we have*

$$\frac{\lambda_1(\mathcal{L})}{\|\mathbf{b}_i^*\|} \le i^{\frac{1+\log i}{2}}. \tag{3}$$

With this lemma in consideration, we can prove the theorem that follows.

Theorem 1. *Let $1 < \beta < n$. For a β-blockwise Korkin-Zolotarev basis $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ and its Gram-Schmidt orthogonalization is $[\mathbf{b}_1^*, \ldots, \mathbf{b}_n^*]$, we have, for $1 \le i \le \beta$,*

$$\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|} < i^{\frac{1+\log i}{2}}, \tag{4}$$

and, for $\beta < i \le n$,

$$\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|} \le \beta^{\frac{1+\log\beta}{2}(\frac{i-1}{\beta-1}+1)}. \tag{5}$$

Proof. Since $\mathbf{B} = [\mathbf{b}_1, \ldots, \mathbf{b}_n]$ is a β-Blockwise Korkin-Zolotarev basis, the sequence of its first β vectors $[\mathbf{b}_1, \ldots, \mathbf{b}_\beta]$ constitutes a Korkin-Zolotarev basis. Then, Lemma 1 yields that, for any such sequence, we have, for $1 \le i \le \beta$,

$$\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|} \le i^{\frac{1+\log i}{2}},$$

which proves the Inequality (4).

Likewise, for every $1 \le i \le n$, $[\pi_i(\mathbf{b}_i), \ldots, \pi_i(\mathbf{b}_{\min(n, i+\beta-1)})]$ constitutes a Korkin-Zolotarev basis. Then, Lemma 1 yields that,

$$\frac{\|\mathbf{b}_i^*\|}{\left\|\mathbf{b}_{\min(n,i+\beta-1)}^*\right\|} \le \beta^{\frac{1+\log\beta}{2}}.$$

So, for $\beta < i \le n$,

$$\begin{aligned}\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|} &\le \frac{\|\mathbf{b}_1^*\|}{\left\|\mathbf{b}_\beta^*\right\|} \times \frac{\left\|\mathbf{b}_\beta^*\right\|}{\left\|\mathbf{b}_{2\beta-1}^*\right\|} \times \cdots \times \frac{\left\|\mathbf{b}_{\lfloor\frac{i-1}{\beta-1}\rfloor(\beta-1)+1}^*\right\|}{\|\mathbf{b}_i^*\|} \\ &\le \beta^{\frac{1+\log\beta}{2}\lceil\frac{i-1}{\beta-1}\rceil} \le \beta^{\frac{1+\log\beta}{2}(\frac{i-1}{\beta-1}+1)}.\end{aligned}$$

This completes the proof. □

3.3 y-Sparse Representations of the Short Lattice Vectors

Equation (2) shows that, for $\mathbf{y} \sim \mathbf{v}$ that is the shortest vector in the lattice $\mathcal{L}(\mathbf{B})$, $\sum_{i=1}^{n} |y_i|^2 \left\|b_i^*\right\|^2$ is bounded by four times the squared first minima λ_1. If the lattice basis $\mathbf{B}$ is **LLL** or β-**BKZ** reduced, the absolute values of elements in $\mathbf{y}$ corresponding to a short vector are even smaller, as Theorem 1 shows, and, for example, fixing $\beta = \sqrt{n}$ yields

$$|y_i| \leq 2^{o(n)}. \tag{6}$$

However, the $\mathbf{y}$ corresponding to the short vectors behaves much better than described above. For example, after performing only **LLL**-reduction on basis of random lattice of dimension 50 (seed $= 0$) in the SVP challenge [31], its $\mathbf{y}$ corresponding to the shortest vector $\mathbf{v}$, whose norm $\|\mathbf{v}\|$ is 1893 as posted on the website, has got only 7 nonzero elements and all the absolute value of the non-zero $|y_i| = 1$, as shown below: the shortest vector $\mathbf{v} =$ [-13 -124 -146 277 -107 -180 673 -311 -167 47 200 395 167 -25 -136 -392 117 -165 147 -515 185 637 343 8 247 44 -220 -146 52 135 -347 -369 -332 -102 469 -285 1 167 397 84 -97 -138 -135 218 567 141 72 21 312 -41], and its corresponding $\mathbf{y}$ under the **LLL**-reduced basis is $\mathbf{y} =$ [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 -1 0 0 -1 1 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 -1]. After computing $\mathbf{y}$ corresponding to the short vectors in the random lattices with dimension 50-120 under **LLL**-reduced or β-**BKZ** basis (β is as small as $\sqrt{n}$), we find that $\mathbf{y}$ representations of the short vectors in random lattices with larger dimensions are similarly well-behaved: $\mathbf{y}$s are quite *sparse* for short vectors, i.e. most of the elements in $\mathbf{y}$ are zero and the non-zero elements are of small absolute values, at most 3, for all the random lattices with dimensions less than 120. Similar sparse $\mathbf{y}$s corresponding to short vectors can also be found in the lattice challenge [21] for q-ary lattices.

We call the sparse $\mathbf{y} = (y_1, \ldots, y_n)$ corresponding to the short lattice vectors as *the y-sparse representations.* The genetic algorithm to be described in details in the next section will be based on the y-sparse representation. In addition, the y-sparse representation is of independent importance to the algorithms for Shortest Vector Problem (SVP).

4 A Genetic Algorithm for SVP

4.1 Encoding a Lattice Vector as a Chromesome

Definition 2. *Given a signed integer x, we define the bit string $(x)_2 = \mathbf{sgn}(x) \vdash (|x|)_2$ as x's binary representation with a sign at the leftmost bit. And its length is*

$$|(x)_2| = 1 + (1 + \lfloor \log |x| \rfloor) = 2 + \lfloor \log |x| \rfloor.$$

Still, we can also represent a signed integer into a bit string of a fixed length ℓ as $(x)_2^{\ell} = \mathbf{sgn}(x) \vdash (|x|)_2^{\ell-1}$.

Definition 3. *Given an integer vector $\mathbf{x} = (x_1, x_2, \ldots, x_n)$ including n signed integers, we define the chromesome $\mathbf{ch}(\mathbf{x}) \in \{0,1\}^{\ell}$ with regards to $\mathbf{x}$ as a bit string $(x_1)_2 \vdash (x_2)_2 \vdash \ldots \vdash (x_n)_2 = \mathbf{sgn}(x_1) \vdash (|x_1|)_2 \vdash \mathbf{sgn}(x_2) \vdash (|x_2|)_2 \vdash \ldots \vdash \mathbf{sgn}(x_n) \vdash (|x_n|)_2$. Then, the length ℓ of the chromesome is*

$$\ell = \sum_{i=1}^{n} (1 + |(|x_i|)_2|).$$

We can also represent the integer vector $\mathbf{x}$ as a chromesome of a fixed length as

$$\begin{aligned} \mathbf{ch}_{\ell_1,\ell_2,\ldots,\ell_n}(\mathbf{x}) &= (x_1)_2^{\ell_1} \vdash (x_2)_2^{\ell_2} \vdash \ldots \vdash (x_n)_2^{\ell_n} \\ &= \mathbf{sgn}(x_1) \vdash (|x_1|)_2^{\ell_1-1} \vdash \mathbf{sgn}(x_2) \vdash (|x_2|)_2^{\ell_2-1} \vdash \ldots \vdash \mathbf{sgn}(x_n) \vdash (|x_n|)_2^{\ell_n-1}, \end{aligned} \tag{7}$$

whose length is, clearly, $\ell_1 + \ell_2 + \ldots + \ell_n$.
Finally, we define the reverse transformation from a chromesome $\mathbf{a} \in \{0,1\}^{\ell_1+\ell_2+\ldots+\ell_n}$ back into the integer vector as $\mathbf{ch}^{-1}_{\ell_1,\ell_2,\ldots,\ell_n}(\mathbf{a}) = \mathbf{x}$ if and only if

$$\mathbf{ch}_{\ell_1,\ell_2,\ldots,\ell_n}(\mathbf{x}) = \mathbf{a}.$$

As we will see in the following sections, genetic algorithms can only operate on chromesomes, therefore, we must represent each lattice vector as a sequence of 0 and 1. Given a lattice vector $\mathbf{v} = \mathbf{B}\mathbf{x}$, we can compute $\mathbf{t} = (t_1, \ldots, t_n)$ as

$$t_i = \sum_{j=i+1}^{n} \mu_{ji} \cdot x_j,$$

and $\mathbf{y} = \mathbf{x} + \lfloor \mathbf{t} \rceil$, we encode the chromesome regarding to the lattice point $\mathbf{v}$ using the integer vector $\mathbf{y}$ instead of $\mathbf{x}$. Then, the chromesome of $\mathbf{v}$ is represented as

$$\mathbf{ch}_{\ell_1,\ldots,\ell_n}(\mathbf{y}) = (y_1)_2^{\ell_1} \vdash (y_2)_2^{\ell_2} \vdash \ldots \vdash (y_n)_2^{\ell_n},$$

and the length ℓ of the whole chromesome for lattice vector $\mathbf{v}$ is

$$\ell = \sum_{i=1}^{n} \ell_i.$$

4.2 Description of the Genetic Algorithm with an Elitist Strategy

In this subsection, we describe our genetic algorithm for the shortest vector problem, and we prove the correctness of the algorithm in the following two subsections.

Before describing our algorithm, it is necessary to define some notions.

Definition 4. *Given two chromesomes $\mathbf{a}, \mathbf{b} \in \{0,1\}^\ell$, for an index vector $\mathbf{u} \in \{0,1\}^\ell$, we define*

$$\varphi_{\mathbf{u}}(\mathbf{a}, \mathbf{b}) = (\mathbf{a} \otimes \mathbf{u}) \oplus (\mathbf{b} \otimes \bar{\mathbf{u}}),$$

in which $\otimes$, $\oplus$ are the bitwise AND and XOR operations respectively. For an index vector $\mathbf{m} \in \{0,1\}^\ell$,

$$\psi_{\mathbf{m}}(\mathbf{a}) = \mathbf{a} \oplus \mathbf{m}.$$

Definition 5 (Crossover and Mutation). *Given two chromesomes $\mathbf{a}$ and $\mathbf{b} \in \{0,1\}^\ell$, we define*

$$\mathbf{crossover}(\mathbf{a}, \mathbf{b}) = \varphi_{\mathbf{u}}(\mathbf{a}, \mathbf{b}),$$

where $\mathbf{u} \in \{0,1\}^\ell$ is uniformly distributed, and

$$\mathbf{mutation}(\mathbf{a}) = \psi_{\mathbf{m}}(\mathbf{a}),$$

where $\mathbf{m} \in \{0,1\}^\ell$ is (ℓ, p_m)-binomially distributed. Then, it is clear that

$$(\mathbf{crossover}(\mathbf{a}, \mathbf{b}))_i = \begin{cases} a_i & \text{at a probability of 1/2,} \\ b_i & \text{at a probability of 1/2.} \end{cases}$$

and

$$(\mathbf{mutation}(\mathbf{a}))_i = \begin{cases} \bar{a}_i & \text{at a probability of } p_m, \\ a_i & \text{at a probability of } 1 - p_m. \end{cases}$$

where $1 \le i \le \ell$.

A classical genetic algorithm operates on a number of binary strings $[\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_p]$ of a certain length ℓ, called a *population*, and the number of the population p is called the *population size*, in which each individual ℓ-length bit string is called a *chromesome* (recall that we define). With each chromesome the algorithm endows a real-valued fitness function $f : \{0,1\}^\ell \to \mathbb{R}$ to signify how good a chromesome is. The genetic algorithm randomly generates an initial population of p chromesomes, and performs iterations of genetic operations selection, crossover and mutation to generate a new population as long as the

Table 3. Genetic Algorithm for the Shortest Vector

```
Input: A lattice basis $\mathbf{B} \in \mathbb{R}^{n\times n}$, the block size $\beta$, the mutation rate $p_m$,
  and the population size $p$.
Output: A lattice vector $\mathbf{v}_0$, such that $\|\mathbf{v}_0\| = \lambda_1(\mathcal{L})$.
1. Preprocess the original basis $\mathbf{B}$ with $\beta$-BKZ procedure, generating $\mathbf{B}'$;
2. Compute $\mathbf{B}'$ Gram-Schmidt orthogonalization sequence $\mathbf{B}^* = [\mathbf{b}_1^*, \cdots, \mathbf{b}_n^*]$ and $[\mu_{ij}]_{1\leq i,j\leq n}$;
3. Perform initialization($\mathbf{B}^*, p$) and get $p$ tuples and bit length $\ell_1, \ldots, \ell_n$;
//   get $p$ lattice vectors randomly;
4. Exchange the $p$ tuples in a non-descending order as $[(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_p, \mathbf{y}_p)]$
  according to their Euclidean norms $\|\mathbf{B}\mathbf{x}_i\|$.
5. Let $\mathbf{v}_0 \leftarrow \mathbf{B}\mathbf{x}_1$. While $\|\mathbf{v}_0\| > \lambda_1(\mathcal{L})$ do//$\lambda_1(\mathcal{L})$ can be its Gaussian Heuristic
  (a) Let $k \leftarrow 1$;
  (b) Select $i, j$ from [1,..., $p$] using proportional selection;
  (c) $\mathbf{b} = $ mutation(crossover($\mathbf{ch}_{\ell_1,\ldots,\ell_n}(\mathbf{y}_i), \mathbf{ch}_{\ell_1,\ldots,\ell_n}(\mathbf{y}_j)$));
  (d) Compute $\mathbf{y}'_k = \mathbf{ch}^{-1}_{\ell_1,\ldots,\ell_n}(\mathbf{b})$ and $\mathbf{x}'_k = \mathbf{y}'_k - \lfloor \mathbf{t}'_k \rceil$ and store the tuple $(\mathbf{x}'_k, \mathbf{y}'_k)$;
    If $\mathbf{x}'_k$ is a zero vector, remove $(\mathbf{x}'_k, \mathbf{y}'_k)$ and go to (c);// zero vector is not in consideration.
  (e) Let $k \leftarrow k + 1$, and if $k < p - 1$, go to (b);
  (f) Let $(\mathbf{x}'_p, \mathbf{y}'_p) \leftarrow (\mathbf{x}_1, \mathbf{y}_1)$ and get new $p$ tuples $[(\mathbf{x}'_1, \mathbf{y}'_1), (\mathbf{x}'_2, \mathbf{y}'_2), \ldots, (\mathbf{x}'_p, \mathbf{y}'_p)]$;
    // reserve the best vector in the last generation of population as the elitism strategy
  (g) Exchange them in a non-descending order and store again as $[(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_p, \mathbf{y}_p)]$
    according to their Euclidean norm $\|\mathbf{B}\mathbf{x}_i\|$;
  (h) If $\|\mathbf{v}_0\| \geq \|\mathbf{B}\mathbf{x}_1\|$, let $\mathbf{v}_0 \leftarrow \mathbf{B}\mathbf{x}_1$;
6. Return $\mathbf{v}_0$.
```

Table 4. Initialization

```
Input: $\mathbf{B}^*$, and the population size $p$.
Output: $p$ tuples $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_p, \mathbf{y}_p)$
1. Compute $\alpha_i = \frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|}$ and $\ell_i = 2 + \lfloor \log \alpha_i \rfloor$ for all $1 \leq i \leq n$;
2. For $k \leftarrow 1$ to $p$ do
  (a) Choose uniformly at random $n$ integers $|y_i| \leq \alpha_i$ in which $1 \leq i \leq n$;
  (b) Let $x_i = y_i - \lceil t_i \rfloor$ in which $t_i = \sum_{j=i+1}^{n} \mu_{ji} x_j$ for each $i$;
  (c) Let $\mathbf{x}_k = (x_1, \ldots, x_n)$, and $\mathbf{y}_k = (y_1, \ldots, y_n)$;
  (d) Store the $k^{th}$ tuple $(\mathbf{x}_k, \mathbf{y}_k)$;
3. Return the $p$ tuples $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_p, \mathbf{y}_p)$ and $\ell_1, \ldots, \ell_n$.
```

Table 5. Crossover

```
Input: Two bit strings $\mathbf{a} = (a_1, \ldots, a_\ell), \mathbf{b} = (b_1, \ldots, b_\ell)$ of length $\ell$ bits.
Output: A bit string $\mathbf{c}$ of length $\ell$
1. For $i \leftarrow 1$ to $\ell$ do
  (a) Choose a number $r \in [0, 1)$ uniformly at random;
  (b) If $r < 0.5$, let $c_i \leftarrow a_i$, and, otherwise, $c_i \leftarrow b_i$;
2. Return $\mathbf{c} = (c_1, \ldots, c_\ell)$.
```

algorithm has not found the chromesome with the maximum fitness among all chromesomes, i.e., the optimum solution.

As described in Table 3, the algorithm reproduces new population of new vectors from the current population in each iteration, until we find a shortest vector in the new generation. We call each population produced in an iteration a *generation* of population. In the iteration, we generate $p-1$ new vectors by using the following scheme. Firstly, we choose two vectors in the current population $(\mathbf{x}_i, \mathbf{y}_i)$ and $(\mathbf{x}_j, \mathbf{y}_j)$ $(1 \leq i, j \leq n)$ by the proportional selection based on the fitness function. Secondly, perform $\mathbf{ch}_{\ell_1,\ldots,\ell_n}(\mathbf{y}_i)$ and

Table 6. Mutation

```
Input: One bit strings $\mathbf{a} = (a_1, \ldots, a_\ell)$ of length $\ell$ bits, and the mutation probability $p_m = 1/\ell$.
Output: A bit string $\mathbf{c}$ of length $\ell$ bits.
1. For $i \leftarrow 1$ to $\ell$ do
   (a) Choose a number $r \in [0, 1)$ uniformly at random;
   (b) If $r < 1/\ell$, let $c_i \leftarrow \bar{a}_i$, and, otherwise, $c_i \leftarrow a_i$;//$\bar{a}_i$ is the flip of the bit $a_i$
2. Return $\mathbf{c} = (c_1, \ldots, c_\ell)$.
```

Table 7. $\mathbf{Ch}_{\ell_1,\ldots,\ell_n}$

```
Input: An integer vector $\mathbf{y} = (y_1, \ldots, y_n)$;
Output: A bit string $\mathbf{a}$ of length $\ell = \sum_{i=1}^{n} \ell_i$ bits.
1. Let $\mathbf{a} = \mathbf{sgn}(y_1) \vdash (|y_1|)_2^{\ell_1 - 1} \vdash \mathbf{sgn}(y_2) \vdash (|y_2|)_2^{\ell_2 - 1} \vdash \ldots \vdash \mathbf{sgn}(y_n) \vdash (|y_n|)_2^{\ell_n - 1}$;
   //$(|\cdot|)_2^{\ell}$ means transforming an integer into its binary representation of length exactly $\ell$;
2. Return $\mathbf{a}$.
```

Table 8. $\mathbf{Ch}^{-1}_{\ell_1,\ldots,\ell_n}$

```
Input: A bit string vector $\mathbf{a} = (a_1, \ldots, a_\ell)$ of length $\ell = \sum_{i=1}^{n} \ell_i$;
Output: An integer vector $\mathbf{y} = (y_1, \ldots, y_n)$.
1. Let $\ell \leftarrow 1$; For $i \leftarrow 1$ to $n$ do
   (a) $y_i = \sum_{i=1}^{\ell_i - 1} a_{\ell+i} \cdot 2^i$; // transform a bit string back into a decimal integer
   (b) If $a_\ell$ is 1, $y_i \leftarrow -y_i$; // add the sign
   (c) $\ell \leftarrow \ell + \ell_i$;
2. Return $\mathbf{y} = (y_1, \ldots, y_n)$.
```

$\mathbf{ch}_{\ell_1,\ldots,\ell_n}(\mathbf{y}_j)$ to obtain two chromesomes for the two vectors. Thirdly, perform the subroutine `crossover` and `mutation` on the two chromesomes to gain a new bit string $\mathbf{b} \in \{0, 1\}^{\ell_1+\ell_2+\ldots+\ell_n}$. Finally, we recover the $\mathbf{y} = ch_{\ell_1,\ldots,\ell_n}(\mathbf{b})$ and $\mathbf{x} = \mathbf{y} - \lfloor \mathbf{t} \rceil$ to reconstruct a tuple $(\mathbf{x}, \mathbf{y})$. Thereby, we yield a new vector from two vectors in the current population, if $\mathbf{x}$ is not a zero vector for the reason that the shortest vector problem means to find the nonzero shortest vector in a lattice. Similarly, we can generate independently $p-1$ such tuples (vectors) to constitute a new population. Besides, we reserve the vector with the shortest Euclidean norm in the current population as the p^{th} vector in the new population, which is called *an Elitist Strategy.* The elitist strategy assures the convergence in the finite generations to the optimum population as we will prove in the next section.

In our genetic algorithm, we use the *proportional selection* to select a chromesome from a population based on its fitness as:

$$\Pr[\mathbf{c}_i \text{ is selected}] = \frac{f(\mathbf{c}_i)}{\sum_{j=1}^{p} f(\mathbf{c}_j)}.$$

As far as the shortest lattice vector problem (**SVP**) is concerned, we regard the lattice vectors $\mathbf{v} = \mathbf{Bx}$ as the individuals and, as defined in Section 4.1, we encode the vectors into chromesomes using the **ch** operation based on the vectors $\mathbf{y} = \mathbf{x} + \lfloor \mathbf{t} \rceil$. We define the fitness function f as the inverse of the Euclidean norm of the vector as:

$$f(\mathbf{v}) = \frac{1}{\|\mathbf{v}\|^2} = \frac{1}{\|\mathbf{Bx}\|^2}.$$

Thereby, the shortest vector obtains the maximum fitness, and enjoys the largest probability that it is selected in the procedure of proportional selection. We choose the square of the Euclidean norm as the fitness function to avoid computing the square root, which is time-assuming in the program implementation.

The optimum of the mutation probability in genetic algorithms is proved to be $1/\ell$ by Bäck [3]. Therefore, the probability that mutation generates the solution a' from individual a is given by,

$$\Pr(a \xrightarrow{mutation} a') = p_m^{Ham(a,a')}(1-p_m)^{(\ell - Ham(a,a'))} > 0,$$

which is very important to ensure the convergence of the algorithm, and here $Ham(a, a')$ denotes the hamming distance of two chromesomes a and a'.

4.3 Convergence of the Elitist Genetic Algorithm

In this section, we prove that the genetic algorithm with an elitist strategy converges to the optimum populations if it iterates a large enough number of generations, under a Markov analysis. For more about Markov stochastic processes, you can refer to [37].

First, denote $\mathcal{N}$ be the set of all the $N = 2^\ell$ lattice vectors. Define $\mathcal{C} = \{\|v\|, v \in \mathcal{N}\}$, suppose that $\mathcal{C}$ has s elements $\{F_1, F_2, \ldots, F_s\}$, where $F_1 < F_2 < \cdots < F_s$. This immediately gives us a opportunity to partition $\mathcal{N}$ into a collection of s nonempty subsets $\mathcal{N}_i$, where

$$\mathcal{N}_i = \{v \in \mathcal{N}, \|v\| = F_i\}.$$

Then for a population P, define

$$\|P\| = \min_{\mathbf{v} \in P} \|\mathbf{v}\|.$$

Denote $\mathcal{P}$ be the set of all the populations. Thus we can consider the collection of populations of the same norm as a set, denoted by E_i, $i = 1, 2, \ldots, s$, that is

$$E_i = \{P \in \mathcal{P}, \|P\| = F_i\}.$$

Let $e_i (> 0)$ be the number of populations in E_i, and P_{ij} be the jth population of E_i, $j = 1, 2, \ldots, e_i$ and $i = 1, 2, \ldots, s$. In other words, any population in $\mathcal{P}$ is represented as P_{ij} for some i and j.

In any generation, the genetic operators create a population P_{kl} from some population P_{ij}. As in our strategy we always preserve the previous best in the population, then the norm of the new generation is at most F_i, thus the genetic operators cannot generate a population P_{kl} from P_{ij}, if $k > i$. The creation of a population P_{kl} from P_{ij} can be viewed as a transition from P_{ij} to P_{kl}. Denote $p_{ij \to kl}$ be the probability that the genetic operators result in the population $P_{kl} \in E_k$ from $P_{ij} \in E_i$, and $p_{ij \to k}$ be the probability of transition from P_{ij} to an population in E_k. Obviously,

$$p_{ij \to k} = \sum_{l=1}^{e_k} p_{ij \to kl}, \text{ and } \sum_{k=1}^{s} p_{ij \to k} = 1.$$

Therefore, we can get the following conclusion.

Theorem 2. *For all $j = 1, 2, \ldots, e_i$, and $i = 1, 2, \ldots, s$,*

$$p_{ij \to k} \begin{cases} > 0, \text{ if } k \le i \\ = 0, \text{ otherwise.} \end{cases}$$

Proof. Let $P = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_p\}$ be a population generated using the genetic operators on P_{ij}. As the best chromosome $\mathbf{v}_0 \in P_{ij}$ is copied in the population P, then $\|P\| \le \|\mathbf{v}_0\| = F_i$. This implies $p_{ij \to kl} = 0$ for all $l = 1, 2, \ldots, e_k$, if $k > i$, which implies $p_{ij \to k} = 0$ directly.

Now for $k \le i$, consider a population P_{kl_1} containing the best chromosome $\mathbf{v}_0$ in P_{ij}. It is clear that the probability will be minimum if we need to mutate each character of each chromosome present in P_{ij} after the selection and crossover operations. As for our strategy, we copy the best chromosome into the new generation, there are at most $(p-1)l$ characters need to mutate, then the minimum probability to obtain P_{kl_1} from P_{ij} is $\left(\frac{1}{l}\right)^{(p-1)l} > 0$, then $p_{ij \to k} = \sum_{l=1}^{e_k} p_{ij \to kl} \ge p_{ij \to kl_1} > 0$, for all $k \le i$. □

One can consider any population P_{ij} as a state of a Markov chain. The probabilities $p_{ij\to kl}$ represent the transition probabilities from population P_{ij} to P_{kl}. In order to denote the transition probability matrix Pr for this Markov chain, let us order the states as $P_{ll}, \dots, P_{1e_1}, P_{21}, \dots, P_{2e_2}, \dots, P_{s1}, \dots, P_{se_s}$. Let $p^{(n)}_{ij\to kl}$ be the probability that GA results in P_{kl} at the nth step given the initial state is P_{ij}, then the $n-$step transition probability matrix $Pr^{(n)} = Pr^n$. Let $p^{(n)}_{ij\to k} = \sum\limits_l p^{(n)}_{ij\to kl}$, then $p^{(n)}_{ij\to k}$ denotes the probability of reaching a population in E_k in n steps with the starting population as P_{ij}. Therefore, as a direct consequence of Theorem 2 in [4], we have the following result to guarantee the convergence of our algorithm.

Theorem 3. *In our GA strategy, for a mutation probability* $\frac{1}{l}$,
$\lim\limits_{n\to\infty} p^{(n)}_{ij\to k} = 0$, *for* $2 \le k \le s$, $\forall j = 1, 2, \dots, e_i$ *and* $i = 1, 2, \dots, s$.
Hence, $\lim\limits_{n\to\infty} p^{(n)}_{ij\to 1} = 1$, $\forall j = 1, 2, \dots, e_i$ *and* $i = 1, 2, \dots, s$.

Note that E_1 represents the set of populations containing the shortest vector, thus Theorem 3 shows that the genetic algorithm will return the shortest vector if it runs long enough.

5 Further Improvements on Our Genetic Algorithm

In this section, we adopt some heuristic techniques to the genetic algorithm to improve its efficiency: the local search and heuristic pruning, which we will discuss in details.

5.1 Local Search

Table 9. Local Search

```
Input: $\mathbf{B}$, and an integer vector $\mathbf{y} = (y_1, \dots, y_n)$;
Output: An integer vector $\mathbf{y}'' = (y''_1, \dots, y''_n)$.
1. Let $\mathbf{y}'' \leftarrow \mathbf{y}$, and compute $\mathbf{v}''$ such that $\mathbf{y}'' \sim \mathbf{v}''$;
2. $p \leftarrow 1$;
3. While $p = 1$ do // if $\mathbf{y}'$ has been updated in the last iteration.
   (a) Compute $\mathbf{v}$ such that $\mathbf{y} \sim \mathbf{v}$;
   (b) Let $p \leftarrow 0$. For all $\mathbf{y}'$ such that $\|\mathbf{y}' - \mathbf{y}\| = 1$ do
      (i) Compute $\mathbf{v}'$ such that $\mathbf{y}' \sim \mathbf{v}'$;
      (ii) If $0 < \|\mathbf{v}''\| < \|\mathbf{v}'\|$, then let $\mathbf{y}'' \leftarrow \mathbf{y}'$, $\mathbf{v}'' \leftarrow \mathbf{v}'$ and $p \leftarrow 1$;
   (c) Let $\mathbf{y} \leftarrow \mathbf{y}''$; //Choose the best $\mathbf{y}''$ nearby $\mathbf{y}$;
4. Return $\mathbf{y}'' = (y''_1, \dots, y''_n)$.
```

Local search is a kind of hill-climbing algorithm which is capable of finding the locally optimal point nearby a given point. As shown in Table 9, given an integer vector $\mathbf{y}$ which represents a specific vector $\mathbf{v}$ in lattice $\mathcal{L}(\mathbf{B})$, we first choose the lattice vector $\mathbf{v} \sim \mathbf{y}$ as the critical vector which serves as the starting point from which we search all the nearby vectors. Starting from Step (b), the algorithm searches all the vectors $\mathbf{y}'$ in $2n$ directions with distance 1, i.e. $\|\mathbf{y}' - \mathbf{y}\| = 1$, and finds the best nearby vector $\mathbf{y}''$ whose corresponding lattice vector $\mathbf{v}'' \sim \mathbf{y}''$ is shorter than the critical vector $\mathbf{v}$. Then the algorithm chooses $\mathbf{v}''$ as the new critical vector, and repeats the steps above until all the $2n$ nearby vectors are no better than the critical vector. For short, local search always takes the steepest direction. Note that local search can always find a vector which is at least as short as the given vector, and that is a locally optimal vector near the given one. Local search may sometimes not find the globally optimal vector, i.e. the shortest vector, in the sense of "greedy" algorithm, but it can never miss it if the given vector is near the shortest vector. So it enhances the probability that it succeeds in finding the shortest vector for a random search algorithm.

In the genetic algorithm, we perform local search on all the newly-generated vectors after performing `crossover` and `mutation`, and we can find a locally optimal vector nearby. Experiments in the next section show that local search helps reduce the running time greatly.

5.2 Optimizations: Heuristic Pruning

As in the y-sparse representation of the shortest vector of the random lattice with dimension 50 under a **LLL**-reduced basis, $\mathbf{y}$ =[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 -1 0 0 -1 1 0 0 0 0 0 1 0 0 0 0 0 -1 0 0 0 -1], we can observe that: first, the first half of the elements of $\mathbf{y}$ are all zeros; second, all the nonzero elements are quite small. Experiments show that the two observations hold in the y-sparse representation of short vectors in the random lattice with dimensions 50-120 under a **LLL**- or **BKZ**-reduced basis. Therefore, in order to obtain short vectors, we can adopt the two heuristic techniques below.

Technique 1: We fix the first half of the elements y_i, for $1 \le i \le \lfloor n/2 \rfloor$, of $\mathbf{y}$ as 0 beforehand.

Technique 2: We bound the rest elements y_i, for $\lfloor n/2 \rfloor \le i \le n$, of $\mathbf{y}$ as $\alpha_i = \sqrt{\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|}}$ instead of $\alpha_i = \frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|}$.

The two techniques help narrow the search space dramatically, like the pruning techniques in the enumeration algorithms. So we call the two techniques as *Heuristic Pruning*. Albeit heuristically, the two pruning never miss any shortest vectors in dimensions 50-120 as the experiments show below. Adopting the two heuristic techniques, we modify the **initialization** procedure as heuristic initialization shown in Table 10 as in Step (1) and Step (2). In fact, we can try much smaller α_i than the suggested in Table 10.

Table 10. Heuristic Initialization

```
Input: $\mathbf{B}^*$, and the population size $p$.
Output: $p$ tuples $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_p, \mathbf{y}_p)$
1. Let $\ell_i \leftarrow 0$ for $1 \le i \le \lfloor n/2 \rfloor$;
2. Compute $\alpha_i = \sqrt{\frac{\|\mathbf{b}_1^*\|}{\|\mathbf{b}_i^*\|}}$ and $\ell_i = 2 + \lfloor \log \alpha_i \rfloor$ for all $\lfloor n/2 \rfloor < i \le n$;
3. For $k \leftarrow 1$ to $p$ do
   (a) Choose uniformly at random $n$ integers $|y_i| \le \alpha_i$ in which $1 \le i \le n$;
   (b) Let $x_i = y_i - \lceil t_i \rfloor$ in which $t_i = \sum_{j=i+1}^{n} \mu_{ji} x_j$ for each $i$;
   (c) Let $\mathbf{x}_k = (x_1, \ldots, x_n)$, and $\mathbf{y}_k = (y_1, \ldots, y_n)$;
   (d) Store the $k^{th}$ tuple $(\mathbf{x}_k, \mathbf{y}_k)$;
4. Return The $p$ tuples $(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_p, \mathbf{y}_p)$ and $\ell_1, \ldots, \ell_n$.
```

5.3 Parallelization

The main body of our genetic algorithm can be easily parallelized into a concurrent algorithm to reduce the running time. As shown in 1, the genetic algorithm can be implemented in a message-passing methods by using p processes: Process 0 keeps a chromesome pool of this generation, or colony, including p chromesomes (or vectors), and, in each iteration, it distributes randomly two chromesomes to each of the other $p-1$ processes; Process $i(1 \le i \le p-1)$ receives two chromesomes and performs in sequence `crossover`, `mutation`, and `local search`, thereby generating a new chromesome (or vector), and, then, sends the newly-generated vector back to Process 0; finally, Process 0 receives all the newly-generated vectors from all the other $p-1$ processes, as well as the best chromesome in the last generation, and the p chromesomes constitutes a new colony, which ends the current iteration. After that, Process 0 starts the iteration again by distributes chromesomes in the new colony to other processes, and all the p processes

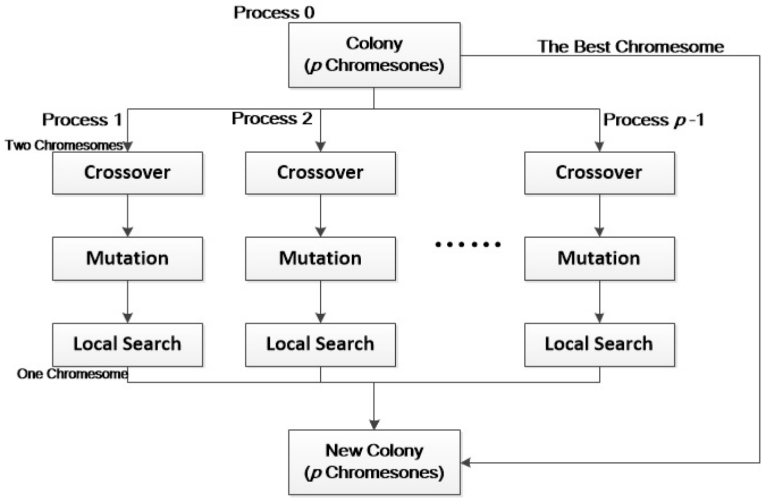

Fig. 1. Parallelization of the Genetic Algorithm

performs the same operations iterations after iterations until Process 0 obtains the shortest vector in its newly-created chromesome pool. Clearly, the parallel program shortens the running time by p times.

Moreover, we can move further to the parallelization. It is easy to keep a number, say n, of concurrent genetic algorithms as described above running in parallel under distinct equivalent lattice basis and using different random seeds, and, therefore, the whole parallel algorithm requires np processes in all. Experiments show that this new parallel algorithm finds the shortest vector amazingly sooner than the sequential genetic algorithm. The parallel program terminates once one of the n concurrent genetic algorithm find the shortest vector.

6 Experimental Results

Although we prove that the genetic algorithm with an elitist strategy is doomed to converge to the shortest vector, we fail to estimate its time complexity (it is still an open problem to prove the first hitting time of the genetic algorithm, except for some special circumstances). Furthermore, the genetic algorithm adopting local search and heuristic pruning is based on the scheme in the sense of greedy strategy and some heuristic observations. So it is necessary to perform experiments to see if it is better than the previous algorithms.

Experiments are performed on a workstation with 16 Intel Xeon 2.4Ghz CPUs and 16G RAM under a Red Hat Linux Server release 5.4. Our genetic algorithm is implemented in C++ using Victor Shoup's Number Theory Library version 6.0.0 [35]. The genetic algorithm chooses the mutation rate $p_m = 1/\ell$, and the population size as twice the rank of the lattice, i.e., $p = 2n$. All the genetic algorithms start on a basis which is $\sqrt{n}$-BKZ reduced, and the basis and its negative vectors serve as the initial population. All the experiments are performed on the random lattices in the SVP challenge [31], which are of extremely large volume (determinant). All the results of lattice with a dimension larger than 91 are obtained using parallel programs as in Section 5.3, and the parallel programs are implemented using MPI, which runs using one thousand cores on a high-performance cluster. The running time of the parallel program are recalculated into that of sequential CPU core programs.

The first experiment means to justify our local search and heuristic techniques, and it runs on the random lattice of dimension 40 generated using seed 0. As shown in Table 11, the genetic algorithm without heuristic techniques finds the shortest vector of norm 1702 using 255.5810 seconds, and the genetic algorithm with local search finds the solution within only 21.8347 seconds. Adopting the two heuristic techniques, the algorithm runs even faster, only 11.8442 seconds. Therefore, we conclude that

Running Time	Heuristic Techniques Adopted
255.5810s	Genetic Algorithm without Heuristic Techniques
21.8347s	Genetic Algorithm with Local Search
11.8442s	Genetic Algorithm with Local Search and Techniques 1 and 2

Table 11. Running Times (Dimension 40) with and without Heuristic Techniques

local search and the heuristic pruning do make sense. All the following experiments are performed using the genetic algorithms with both local search and heuristic pruning.

Dimension	Seed	Previous Challenge	Our Results
40	0	1702	1702
50	0	1893	1893
51	0	1885	1885
66	3	2099	2036
67	0	n/a	2187
69	0	2226	2212
73	0	2220	2190
79	0	n/a	2346
81	0	n/a	2276
83	0	n/a	2433
85	0	n/a	2412
86	0	2456	2387
87	0	n/a	2497
91	0	n/a	2468

Table 12. Comparison of Results (Euclidean Norm) on SVP challenge

Second, we perform experiments using the improved genetic algorithm on the random lattices of dimensions 40-118 generated using mostly seed 0 except for seed 3 of dimension 66. As shown in Table 12, it shows that our genetic algorithm can always find vectors at least as good as previous challenges in the same lattice, some of them even shorter. For example, the previous challenge of lattice of dimension 66 seed 3 is of norm 2099, while we obtain a vector of norm 2036. The previous challenge for dimension 69 is 2226, and our result is 2212, and for dimension 73 our result is 2190 compared to 2220 of previous challenge, and for dimension 86 ours is 2387 as for previous 2456. We also make some records without any previous challenge results as in the table 11, like the dimension 91, 87, 85, and so on. So far, we have posted 25 challenge results (with the largest dimension 91, most in Feb. 2013) on the SVP challenge website (See link [31]), and we have performed experiments on lattices up to dimension 118. We are still performing more experiments using our genetic algorithm on random lattices with even larger dimensions in SVP challenge. Moreover, we have attempted 200-dimensional q-ary lattices of lattice challenge[31].

Finally, we compare our improved genetic algorithm with Kannan-Helftich enumeration and Enumeration with conservative Pruning, which are of polynomial space complexity as the genetic algorithm is. The enumeration with extreme pruning [11], though efficient, suffers from a low rate of success, or, in other words, fails to find a short vector at a high probability, and, then, we choose to implement an enumeration with a relatively conservative pruning, which guarantees finding the shortest vector. Both algorithms are implemented using the same NTL library as our genetic algorithm, and both run on the same 16-CPU workstation. In order to obtain a fair comparison, we preprocess the three algorithms with a $\sqrt{n}$-**BKZ** reduction. The running times of the three algorithms are compared in the Fig. 2. The experiments are performed on random lattices from dimension 20 to 118 generated with seed 0. As shown in the Table, the genetic algorithm outperforms the other two algorithms in running times over all the

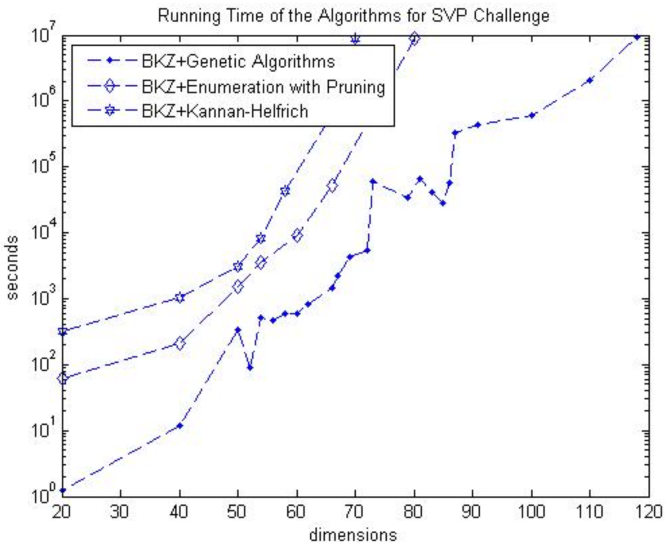

Fig. 2. Comparison of the Logarithms of the Running Times of Main SVP Algorithms

dimensions. The Kannan-Helfrich enumeration runs the slowest among the three, and it succeeds in finding the shortest vector only up to dimension 70. The enumeration with pruning, which we implement only the conservative pruning to make sure the shortest vector can always be found, runs faster than Kannan-Helfrich enumeration and can only find the shortest vector at most dimension 80. As for the genetic algorithm, its performance is much better than the previous two greatly. The genetic algorithm manages to find the shortest vectors for random lattices of up to dimension 118.

7 Conclusion

In this paper, we are the first to propose a genetic algorithm aiming at searching the shortest vector of the random lattices from the SVP challenge [31], which has attracted numerous attention in cryptography community. Our algorithm can converge to the shortest vector at a probability close to 1, and the experimental results show that it is efficient and gains some advantages over other algorithms.

Although only three distinct algorithm descriptions are available to choose from on the website: "enumeration", "**BKZ**", and "sieve", as well as "other", the SVP challenge was opened in 2010 to evaluate and compare all sorts of SVP algorithms. So, it is quite inspiring to post challenge results from novel original algorithms, not just of the three types above. Our algorithm is beyond the three, and, namely, a main body of genetic algorithm with a preprocessing of a **BKZ** reduction, which helps reduce the population space (like reduced Markov chain[26]).

Furthermore, we also attempt some low-dimensional q-ary lattices from the lattice challenge[21] using our genetic algorithm. Although our algorithm only works well on dimension 200, compared to the records of dimension up to 800, it is, we believe, promising for our genetic algorithm to challenge some high-dimensional q-ary lattices, if we find some new heuristic techniques effective for q-ary lattices, like local search and heuristic pruning for the SVP challenge in this paper.

References

1. Ajtai, M. The shortest vector problem in ℓ_2 is np-hard for randomized reductions. *Proceeding of the* 30^{th} *Symposium on the Theory of Computing (STOC 1998)*, 284-406 (1998).
2. Ajtai, M., Kumar, R., and Sivaumar, D. A sieve algorithm for the shortest lattice vector problem. *Proceedings of the* 33^{th} *annual ACM symposium on Theory of computing (STOC'01) 33*, 601-610 (2001).

3. Bäck, T. Optimal mutation rates in genetic search. *Proceedings of the fifth International Conference on Genetic Algorithms 5*, 2-8 (1993).
4. Bhandari, D., Murthy, C., and Pal, S. K. Genetic algorithm with elitist model and its convergence. *International Journal of Pattern Recognition and Artificial Intelligence 10*, 06 (1996), 731–747.
5. Booker, L. B., Goldberg, D. E., and Holland, J. H. Classifier systems and genetic algorithms. *Artificial intelligence 40*, 1 (1989), 235–282.
6. Chen, Y., and Nyuyen, P. Q. Bkz2.0: Better lattice security estimates. *Advances in Cryptology C ASIACRYPT 2011, Lecture Notes in Computer Science 7073*, 1-20 (2011).
7. Chu, P. C., and Beasley, J. E. A genetic algorithm for the multidimensional knapsack problem. *Journal of heuristics 4*, 1 (1998), 63–86.
8. Davis, L. *Handbook of genetic algorithms.* Van Nostrand Reinhold, 1991.
9. Eiben, A. E., Aarts, E. H., and Van Hee, K. M. Global convergence of genetic algorithms: A markov chain analysis. In *Parallel problem solving from nature.* Springer, 1991, pp. 3–12.
10. Fogel, D. B. *Evolving artificial intelligence.* University of California at San Diego, 1992.
11. Gamma, N., Nguyen, P. Q., and Regev, O. Lattice enumeration using extreme pruning. *Advances in Cryptology C EUROCRYPT 2010, Lecture Notes in Computer Science 6110*, 257-278 (2010).
12. Goldberg, D. E., and Holland, J. H. Genetic algorithms and machine learning. *Machine learning 3*, 2 (1988), 95–99.
13. Hanrot, G., and Stehlé, D. Analyzing blockwise lattice algorithms using dynamical systems. *Advances in Cryptology-CRYPTO 4622*, 170-186 (2007).
14. Helfrich, B. Algorithms to construct minkowski reduced and hermit reduced bases. *Theoretical Computer Sciences 41*, 125-139 (1985).
15. Holland, J. H. *Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press, 1975.
16. Kannan, R. Improved algorithms for integer programming and related lattice problems. *Proc. of the* 15th *Symposium on the Theory of Computing (STOC1983) 15*, 99-108 (1983).
17. Kannan, R. Minkowski's convex body theorem and integer programming. *Mathemaatics of Operations Research 12*, 415-440 (1987).
18. Khuri, S., Bäck, T., and Heitkötter, J. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of the 1994 ACM symposium on Applied computing* (1994), ACM, pp. 188–193.
19. Lagrias, J. C., Lenstra, H. W., and Schnorr, C. P. Korkin-zolotarev basis and successive minima of a lattice and its reciprocal lattice. *Combinatorica 10(4)*, 333-348 (1990).
20. Lenstra, A. K., Lenstra, H. W., and Lavász, L. Factoring polynomials with rational coefficients. *Mathematische Annalen 261*, 513-534 (1982).
21. Lindner, R., Rueckert, M., Baumann, P., and Nobach, L. http://www.latticechallenge.org/.
22. Liu, X., Han, W., and Quan, J. A new lattice reduction algorithm based on genetic strategy(chinese version). *Journal of Electronics & Information Technology 35*, 8 (2013), 1940–1945.
23. Micciancio, D., and Goldwasser, S. *Complexity of Lattice Problems: A Cryptographic Perspective.* Kluwer Academic Publishers, 2002.
24. Micciancio, D., and Regev, O. Lattice-based cryptography. *Proceeding of the Post-Quantum Cryptography (PQC'09)*, 147-191 (2009).
25. Micciancio, D., and Voulgaris, P. A deterministic single exponential time algorithm for most lattice problems based on voronoi cell computations. *Proceedings of the* 42th *annual ACM symposium on Theory of computing (STOC'10) 42*, 351-358 (2010).
26. Moey, C. C., and Rowe, J. E. A reduced markov model of gas without the exact transition matrix. In *Parallel Problem Solving from Nature-PPSN VIII* (2004), Springer, pp. 72–80.
27. Mühlenbein, H. Parallel genetic algorithms, population genetics and combinatorial optimization. In *Parallelism, Learning, Evolution.* Springer, 1991, pp. 398–406.
28. Nguyen, P. Q., and Vidick, T. Sieve algorithms for the shortest vector problem are practical. *Journal of Mathematical Cryptography 2*, 2(181C207) (2008).
29. Papadimitriou, C. H., and Steiglitz, K. *Combinatorial optimization: algorithms and complexity.* Courier Dover Publications, 1998.
30. Rudolph, G. Convergence analysis of canonical genetic algorithms. *Neural Networks, IEEE Transactions on 5*, 1 (1994), 96–101.
31. Schneider, M., and Gamma, N. http://www.latticechallenge.org/svp-challenge/.

32. Schnorr, C. P. A hierarchy of polynomial lattice basis reduction algorithms. *Theoretical Computer Sciences 53*, 201-224 (1987).
33. Schnorr, C. P., and Euchner, M. Lattice basis reduction: Improved practical algorithms and solving subset sum problems. *Mathematical Programming 66*, 181-199 (1994).
34. Shor, P. W. Algorithms for quantum computation: discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on* (1994), IEEE, pp. 124–134.
35. Shoup, V. Number theory c++ library (ntl) vesion 6.0.0, available at http://www.shoup.net/ntl/.
36. Srinivas, M., and Patnaik, L. M. Genetic algorithms: A survey. *Computer 27*, 6 (1994), 17–26.
37. Stroock, D. W. *An Introduction to Markov Processes.* Springer Publishers, 2005.
38. Suzuki, J. A markov chain analysis on simple genetic algorithms. *Systems, Man and Cybernetics, IEEE Transactions on 25*, 4 (1995), 655–659.
39. Thierens, D., and Goldberg, D. Convergence models of genetic algorithm selection schemes. In *Parallel problem solving from naturePPSN III.* Springer, 1994, pp. 119–129.
40. van Emde Boas, P. Another np-complete partition problem and the complexity of computing short vectors in a lattice. *Technical Report, Mathematisch Instituut, Universiteit van Amsterdam 81-04* (1981).
41. Wang, X., Liu, M., Tian, C., and Bi, J. Improved nguyen-vidick heuristic sieve algorithm for shortest vector problem. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security* (2011), ACM, pp. 1–9.